

Extending Büchi Automata with Constraints on Data Values*

Ahmet Kara

Technical University of Dortmund

Email: `ahmet.kara@cs.tu-dortmund.de`

Tony Tan

University of Edinburgh

Email: `ttan@inf.ed.ac.uk`

Abstract

Recently data trees and data words have received considerable amount of attention in connection with XML reasoning and system verification. These are trees or words that, in addition to labels from a finite alphabet, carry data values from an infinite alphabet (data). In general it is rather hard to obtain logics for data words and trees that are sufficiently expressive, but still have reasonable complexity for the satisfiability problem. In this paper we extend and study the notion of Büchi automata for ω -words with data. We prove that the emptiness problem for such extension is decidable in elementary complexity. We then apply our result to show the decidability of two kinds of logics for ω -words with data: the two-variable fragment of first-order logic and some extensions of classical linear temporal logic for ω -words with data.

1 Introduction

The classical theory of automata and formal languages deals primarily with languages over finite alphabets. A natural extension of formal languages, regular or context-free, is one that permits the alphabet to be infinite [1, 3, 2,

*We acknowledge the financial support by the European FET-Open Project FoX (grant agreement 233599) and the German DFG (grant SCHW 678/4-1).

5, 13, 14, 17]. Most of the extensions, however, lack the usual nice decidability properties of automata over finite alphabets, unless strong restrictions are imposed.

Recently the subject of languages over infinite alphabets received much attention due to its connection with XML reasoning and system specification. The most natural model for XML documents is label unranked trees, in which each node has a label from a finite alphabet. Thus, standard technique in automata theory can be applied [15, 16, 18]. However, real XML documents carry *data*, which usually come from an infinite set, and it is essential to reason about those data values. Thus, there is a need to look for *decidable formalism* in the presence of a second, infinite alphabet.

A similar scenario may happen in system specification where ω -words (words of infinite length) are used to describe system behaviors. In this case a position in the word represents a point in time, while the label of the position indicates the atomic propositions that hold at that time. The number of atomic propositions is usually only finitely many, and thus, can be encoded as finite alphabets. The most common tool for reasoning with ω -word is arguably Büchi automata, due to its expressiveness and the low complexities for its standard decision problems. For example, it captures the so-called monadic second order (MSO) logic, and hence the specification languages such as Linear Temporal Logic (LTL) and μ -calculus. However, the behaviour of many systems includes properties that cannot be captured by finite alphabets. A typical example is reasoning about the contents of variables, that store values from the infinite domains like the integers or strings. Thus, it is also natural to look for some formalisms that allow us to reason about ω -words with data values that come from an infinite domain.

Our focus in this paper is *data ω -word*, that is, ω -words in which each position also carries a data value from an infinite alphabet. Looking at the literature [2, 3, 4, 8, 9, 10, 13, 14, 16, 17] one can immediately notice that decidable formalisms for data ω -words are hard to obtain, unless strong restrictions are imposed. Nevertheless, some significant progress have been made recently [3, 10, 9]. A deep result in [3] shows that the restriction of first-order logic to its two variable fragment, FO^2 , remains decidable over data ω -words. The pioneering works in Linear Temporal Logic for ω -words with data are the papers [10, 9]. In [9] an extension of Linear Temporal Logic (LTL) to handle data values is proposed and its satisfiability problem is shown to be decidable. In papers [3, 9] the satisfiability problem, even though is decidable, has unknown upper bound complexity. The decidability is obtained by reducing the satisfiability problem to the reachability problem in Petri nets, the precise complexity of which has been open for many

years, though it is known to be in **EXPSpace**-hard. In the paper [10] the logic is decidable, but not primitive recursive, for finite data words, while it becomes undecidable for ω -words. The paper [9] also contains a logic which is decidable in **PSpace**. However, the logic has quite limited expressive power, in which the finite alphabet for the labels consists of only one single symbol.

In this paper we propose and study an extension of Büchi automata with a formalism to specify constraints on data values. Roughly those constraints are database theory inspired, called *key*-, *inclusion*- and *denial*-constraints. A key-constraint states that no two positions labeled with the same symbol a has the same data value; inclusion-constraint states that every data value found in a position with label a is found in a position with label b ; while denial-constraint states that the sets of data values found in positions with labels a and b are disjoint. Those constraints are very common in database theory. We show that the emptiness problem for such extension is decidable in **NEXPTIME**, whereas if there is no key-constraint, then the complexity drops to **NP**. We then apply our results to show the decidability of two kinds of logics for data ω -words: the two-variable fragment of first-order logic and some extensions of classical linear temporal logic for data ω -words. Both have elementary complexity.

The vocabulary for the two-variable logic that we consider here has only the *successor* relation on the positions in the ω -word and the *data equality*, in addition to the finite number of unary predicates for the finite labeling. In [3] the vocabulary includes the *order* on the positions in the ω -words and as mentioned earlier, the satisfiability problem for the two-variable logic becomes at least as hard as the reachability problem for Petri nets.

Another work that is related to our work is the remarkable result in [2], which shows that for two-variable fragment of first-order logic over *finite unranked data trees*, with vocabulary consists of successor and data equality, is decidable in **3-NEXPTIME**. Another proof with different approach for the restricted case of *finite data words* was later obtained in [8].

The paper is organized as follows. In Section 2 we define the notations and tools that we are going to use in this paper. In Section 3 we introduce the extension of Büchi automata by equipping it with data-constraints and we prove that the emptiness problem is decidable in elementary complexity. We call this model *Büchi automata with data-constraints* (ADC). In Section 4 we further extend ADC with operators for comparing the equality between neighboring data values, which we call *profile Büchi automata with data-constraints*. The emptiness problem for this model is also decidable in elementary complexity. Then in Section 5 we present a decision procedure for the satisfiability problem of the two-variable fragment of first-order

logic. Finally in Section 6 we introduce a version of Linear Temporal Logic (LTL) that is equipped with some operators for data value comparisons. For this also we prove that the satisfiability problem is decidable in elementary complexity.

Acknowledgement We thank Claire David, Leonid Libkin and Thomas Schwentick for fruitful discussions.

2 Notations

2.1 Data words

Let Σ be a finite alphabet and \mathfrak{D} an infinite set of data values. A *finite* word is an element of Σ^* , while an ω -word is an element of Σ^ω . A finite *data word* is an element of $(\Sigma \times \mathfrak{D})^*$, while a *data ω -word* is an element of $(\Sigma \times \mathfrak{D})^\omega$.

We write a data (finite or ω -) word w as $\binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$, where $a_1, a_2, \dots \in \Sigma$ and $d_1, d_2, \dots \in \mathfrak{D}$. The symbol a_i is the label of position i , while the value d_i is the data value in position i . The projection of w to the alphabet Σ is denoted by $\text{Proj}(w) = a_1 a_2 \cdots$. A position in w is called an a -position, if the label is a . We denote by $V_w(a)$, the set of data values found in a -positions in w , i.e., $V_w(a) = \{d_i \mid a_i = a\}$, for each $a \in \Sigma$. Note that some $V_w(a)$'s may be infinite, while some others finite.

2.2 Data-constraints: constraints on the data values

There are three kinds of data-constraints over the alphabet Σ :

1. *key-constraints*, written in the form: $V(a) \mapsto a$, where $a \in \Sigma$.
2. *inclusion-constraints*, written in the form: $V(a) \subseteq \bigcup_{b \in R} V(b)$, where $a \in \Sigma$, $R \subseteq \Sigma$.
3. *denial-constraints*, written in the form: $V(a) \cap V(b) = \emptyset$, where $a, b \in \Sigma$.

Whether a data word w satisfies a data-constraint C , written as $w \models C$, is defined as follows.

1. $w \models V(a) \mapsto a$, if every two a -positions in w have different data values.
2. $w \models V(a) \subseteq \bigcup_{b \in R} V(b)$, if $V_w(a) \subseteq \bigcup_{b \in R} V_w(b)$.
3. $w \models V(a) \cap V(b) = \emptyset$, if $V_w(a) \cap V_w(b) = \emptyset$.

If \mathcal{C} is a collection of data-constraints, then we write $w \models \mathcal{C}$, if $w \models C$ for all $C \in \mathcal{C}$.

2.3 Transition systems and Büchi automata

A transition system over the alphabet Σ is a tuple $\mathcal{M} = \langle Q, \mu \rangle$, where Q is a finite set of states and $\mu \subseteq Q \times \Sigma \times Q$ is the set of transitions.

A Büchi automaton \mathcal{A} over the alphabet Σ is simply a transition system \mathcal{M} with a designated initial state q_0 and a set $F \subseteq Q$ of final states. In such case, we write $\mathcal{A} = \mathcal{M}_{q_0}^F$ and the system \mathcal{M} is called the *transition system* of \mathcal{A} .

A run of \mathcal{A} on an ω -word $w = a_1 a_2 \dots$ is a sequence $\rho = p_1 p_2 \dots$ of states in Q such that $(q_0, a_1, p_1) \in \mu$ and $(p_i, a_{i+1}, p_{i+1}) \in \mu$, for each $i = 1, 2, \dots$. Note that we exclude the initial state in the run ρ . This is done for our convenience of indexing.

Let $\text{Inf}(\rho)$ denote the set of states that appear infinitely many times in ρ . The run ρ is accepting, if $\text{Inf}(\rho) \cap F \neq \emptyset$. An ω -word $w \in \mathcal{L}(\mathcal{A})$, if there exists an accepting run of \mathcal{A} on w . As usual, $\mathcal{L}(\mathcal{A})$ denotes the set of ω -words accepted by the automaton \mathcal{A} .

2.4 Presburger automata

Existential Presburger formula

Atomic Presburger formulae are of the form: $x_1 + x_2 + \dots + x_n \leq y_1 + \dots + y_m$, or $x_1 + \dots + x_n \leq K$, or $x_1 + \dots + x_n \geq K$, for some constant $K \in \mathbb{N}$. *Existential Presburger formulae* are Presburger formulae of the form $\exists \bar{x} \phi$, where ϕ is a Boolean combination of atomic Presburger formulae.

We will be using Presburger formulae defining Parikh images of words. Let $\Sigma = \{a_1, \dots, a_k\}$ be a finite alphabet, and let $v \in \Sigma^*$ be a finite word. We denote by $\#_v(a_i)$ the number of occurrences of a_i in v . By $\text{Parikh}(v)$ we mean the *Parikh image* of v , i.e., $(\#_v(a_1), \dots, \#_v(a_k))$.

With alphabet letters a_1, \dots, a_k , we associate variables x_{a_1}, \dots, x_{a_k} . A Presburger formula φ with free variables x_{a_1}, \dots, x_{a_k} is said to be a formula over the alphabet $\{a_1, \dots, a_k\}$. A word $v \in \Sigma^*$ satisfies it, written as $v \models \varphi(x_{a_1}, \dots, x_{a_k})$ if and only if $\varphi(\text{Parikh}(v))$ holds.

Presburger automata

A *Presburger automaton* is a pair $(\mathcal{A}_{\text{fin}}, \varphi)$, where \mathcal{A}_{fin} is a finite state automaton for *finite words* and $\varphi(x_{a_1}, \dots, x_{a_k})$ is an existential Presburger

formula over the alphabet Σ . A word w is accepted by $(\mathcal{A}_{\text{fin}}, \varphi)$, denoted by $\mathcal{L}(\mathcal{A}_{\text{fin}}, \varphi)$, if $w \in \mathcal{L}(\mathcal{A}_{\text{fin}})$ and $\varphi(\text{Parikh}(w))$ holds.

Note that as convention, we will use the symbol \mathcal{A}_{fin} for finite state automata that works over *finite* words. We reserve the symbol \mathcal{A} for Büchi automata, which works over ω -words.

As in [2, 8], the following result is the basis for all the decidability results in this paper.

Theorem 1 [19] *The emptiness problem for presburger automata is decidable in NP.*

3 Automata with data-constraints

In this section we extend the definition of Büchi automata with data-constraints over the input alphabet Σ . We then provide a decision procedure for its emptiness problem, from which all other decision procedures in this paper are extended.

Definition 2 An *Automaton with Data-constraints*, or in short ADC, is a pair $(\mathcal{A}, \mathcal{C})$, where \mathcal{A} is a Büchi automaton and \mathcal{C} is a collection of data-constraints over the alphabet Σ .

Let $w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$ be a data ω -word. The ADC $(\mathcal{A}, \mathcal{C})$ accepts the data ω -word w , if $\text{Proj}(w) \in \mathcal{A}$ and $w \models \mathcal{C}$. We denote by $\mathcal{L}(\mathcal{A}, \mathcal{C})$ the language that consists of all the data ω -words accepted by the ADC $(\mathcal{A}, \mathcal{C})$.

We consider the following problem.

| | |
|-----------|---|
| PROBLEM: | OMEGA-SAT-ADC |
| INPUT: | An automaton with data-constraints $(\mathcal{A}, \mathcal{C})$ |
| QUESTION: | Is there an data ω -word $w \in \mathcal{L}(\mathcal{A}, \mathcal{C})$? |

Theorem 3 *The problem SAT-ADC is decidable in NEXPTIME. Moreover, if the collection \mathcal{C} of data constraints does not contain key-constraints, then it is decidable in NP.*

For the proof we first introduce some essential notations in Subsection 3.1, then we outline the NEXPTIME algorithm in Subsection 3.2. The NP algorithm can be found in Appendix C.

Before we start the first proof in this paper, we want to remark the similarities and differences between the technique in this paper and the

one in [3]. The only similarity is that all techniques rely quite heavily on Presburger counting. However, there is a different emphasis in the counting process: in [2] the technique is to count the number of the so called *dog* labels and *sheep* labels (see pp. 35–36 in [2]), where intuitively, the dog labels are used to represent the data values. In this paper the technique involves counting directly the “number” of data values.

3.1 Some notations for the proof of Theorem 3

For a data ω -word w and a non-empty subset $S \subseteq \Sigma$, we denote by

$$[S]_w = \bigcap_{a \in S} V_w(a) \cap \bigcap_{b \notin S} \overline{V_w(b)},$$

where $\overline{V_w(b)}$ denotes the complement of $V_w(b)$, i.e. $\mathfrak{D} - V_w(b)$. It must be noted that the sets $[S]_w$ ’s are disjoint, and for each $a \in \Sigma$, $V_w(a)$ is partitioned into $V_w(a) = \bigcup_{a \in S} [S]_w$. These two properties (disjointness and partition) of $[S]_w$ ’s are very crucial in our decision procedure.

According to the cardinalities of $[S]_w$ ’s, we divide the non-empty subsets $S \subseteq \Sigma$ into three classes:

- $\mathcal{S}_0(w) = \{S \mid [S]_w = \emptyset\}$.
- $\mathcal{S}_{\text{fin}}(w) = \{S \mid [S]_w \text{ is a finite non-empty set}\}$.
- $\mathcal{S}_{\infty}(w) = \{S \mid [S]_w \text{ is an infinite set}\}$.

Proposition 4 [8, Proposition 1] *For every data ω -word w , the following holds.*

1. $w \models V(a) \subseteq \bigcup_{b \in R} V(b)$ if and only if $S \in \mathcal{S}_0(w)$, for all S such that $a \in S$, but $S \cap R = \emptyset$.
2. $w \models V(a) \cap V(b) = \emptyset$ if and only if $S \in \mathcal{S}_0(w)$, for all S such that $a, b \in S$.

Proof. (2) is immediate from the definition of $[S]_w$, while (1) follows from the fact that

$$V_w(a) \subseteq \bigcup_{b \in R} V_w(b) \quad \text{if and only if} \quad V_w(a) \cap \bigcap_{b \in R} \overline{V_w(b)} = \emptyset.$$

□

3.2 The algorithm

Let $(\mathcal{A}, \mathcal{C})$ be the given ADC and $\mathcal{M} = \langle Q, \mu \rangle$ be the transition system, where $\mathcal{A} = \mathcal{M}_{q_0}^F$. Roughly our algorithm to determine whether $\mathcal{L}(\mathcal{A}, \mathcal{C}) = \emptyset$ is as follows.

1. Guess a partition $\mathcal{S}_0, \mathcal{S}_{\text{fin}}, \mathcal{S}_{\infty}$ of the sets $2^\Sigma - \{\emptyset\}$ that respects the following conditions.
 - (C1) If the inclusion-constraint $V(a) \subseteq \bigcup_{b \in R} V(b)$ is in \mathcal{C} , then all the sets S , where $a \in S$ and $S \cap R = \emptyset$, are in \mathcal{S}_0 .
 - (C2) If the denial-constraint $V(a) \cap V(b) = \emptyset$ is in \mathcal{C} , then all the sets S , which contains both a and b , are in \mathcal{S}_0 .

The intended meaning of the guesses $\mathcal{S}_0, \mathcal{S}_{\text{fin}}, \mathcal{S}_{\infty}$ are the sets $\mathcal{S}_0(w)$, $\mathcal{S}_{\text{fin}}(w)$, $\mathcal{S}_{\infty}(w)$, respectively, for some $w \in \mathcal{L}(\mathcal{A}, \mathcal{C})$.

Moreover, Conditions (C1) and (C2) must be respected due to Proposition 4.

2. Construct the following two items, of which the details are provided below.
 - (a) A new alphabet $\tilde{\Sigma}$, which depend on the original alphabet Σ and the sets in \mathcal{S}_{∞}
 - (b) A transition system $\tilde{\mathcal{M}} = \langle \tilde{Q}, \tilde{\mu} \rangle$ over the alphabet $\tilde{\Sigma}$, which depends on the original transition system \mathcal{M} and the sets in \mathcal{S}_{∞} .
3. Non-deterministically choose one state $q \in \tilde{Q}$ and construct the following two items.
 - a Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$, where $\tilde{\mathcal{A}}_{\text{fin}} = \tilde{\mathcal{M}}_{q_0}^{\{q\}}$ and the formula φ depends on the partition $\mathcal{S}_0, \mathcal{S}_{\text{fin}}, \mathcal{S}_{\infty}$ and the constraints in \mathcal{C} ;
 - a Büchi automaton $\tilde{\mathcal{A}}$, which depends on the constraints in \mathcal{C} , the new transition system $\tilde{\mathcal{M}}$, and the sets in \mathcal{S}_{∞} .
4. Test the emptiness of $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ and $\mathcal{L}(\tilde{\mathcal{A}})$.
 Then, $\mathcal{L}(\mathcal{A}, \mathcal{C}) \neq \emptyset$ if and only if $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi) \neq \emptyset$ and $\mathcal{L}(\tilde{\mathcal{A}}) \neq \emptyset$.

In the paragraphs below we will outline the details of Steps (2) and (3). The analysis of the complexity is given in Appendix A.

The proof of the correctness will follow from our claim that $\mathcal{L}(\mathcal{A}, \mathcal{C}) \neq \emptyset$ if and only if there exist some “correct” guesses for $\mathcal{S}_0, \mathcal{S}_{\text{fin}}, \mathcal{S}_{\infty}$ in Step (1)

and the state $q \in \tilde{Q}$ in Step (3) such that $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi) \neq \emptyset$ and $\mathcal{L}(\tilde{\mathcal{A}}) \neq \emptyset$. The details of the proof of the correctness will be given in Appendix B. The main idea of the proof is that from a word $u \in \mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$, we can construct a finite data word w , and from an omega word $v \in \mathcal{L}(\tilde{\mathcal{A}})$, we can construct a data ω -word w' such that $ww' \in \mathcal{L}(\mathcal{A}, \mathcal{C})$.

Constructing the alphabet $\tilde{\Sigma}$ and the transition system $\tilde{\mathcal{M}}$

We define a set $\Sigma(\mathcal{S}_\infty) = \{(a, S) \mid a \in S \text{ and } S \in \mathcal{S}_\infty\}$. Then, the new alphabet $\tilde{\Sigma}$ is $\tilde{\Sigma} = \Sigma \cup \Sigma(\mathcal{S}_\infty)$. The transition system $\tilde{\mathcal{M}} = \langle \tilde{Q}, \tilde{\mu} \rangle$ is defined as $\tilde{Q} = Q$ and $\tilde{\mu} = \mu \cup \{(p, (a, S), q) \mid (p, a, q) \in \mu \text{ and } (a, S) \in \Sigma(\mathcal{S}_\infty)\}$.

Constructing the Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$

Let $q \in \tilde{Q}$ be the state chosen non-deterministically in Step (3). The automaton $\tilde{\mathcal{A}}_{\text{fin}}$ is simply $\mathcal{M}_{q_0}^{\{q\}}$. The Presburger formula φ is defined as follows. Let S_1, \dots, S_m be the enumeration of non-empty subsets of Σ , where $m = 2^{|\Sigma|} - 1$.

The formula φ is of the form $\exists z_{S_1} \cdots \exists z_{S_m} \psi$, where ψ is the following quantifier-free formula:

$$\begin{aligned} \bigwedge_{a \in \Sigma} x_a \geq \sum_{S \ni a} z_S \quad \wedge \quad \bigwedge_{S \in \mathcal{S}_0 \cup \mathcal{S}_\infty} z_S = 0 \\ \wedge \\ \bigwedge_{S \in \mathcal{S}_{\text{fin}}} z_S \geq 1 \quad \wedge \quad \bigwedge_{V(a) \mapsto a \in \mathcal{C}} x_a = \sum_{a \in S} z_S \end{aligned}$$

Note the constructed formula φ does not involve the symbols in $\Sigma(\mathcal{S}_\infty)$.

Constructing the Büchi automaton $\tilde{\mathcal{A}}$

The Büchi automaton $\tilde{\mathcal{A}}$ is simply the intersection of $\tilde{\mathcal{M}}_q^F$ with the automaton that checks the following conditions.

1. Each $(a, S) \in \Sigma(\mathcal{S}_\infty)$ appears infinitely many times.
2. If the key-constraint $V(a) \mapsto a \in \mathcal{C}$, then the symbol a does not appear.

4 Automata with data-constraints and profiles

Given a data word $w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$, the *profile word* of w , denoted by $\text{Profile}(w)$, is the word

$$\text{Profile}(w) = (a_1, (L_1, R_1)), (a_2, (L_2, R_2)), \dots \in (\Sigma \times \{*, \top, \perp\} \times \{*, \top, \perp\})^\omega$$

such that for each position $i = 1, 2, \dots$, the values of L_i and R_i are either \top , or \perp , or $*$. If $L_i = \top$ and $i > 1$, it means that the position on the left, $i - 1$, has the same data value as position i ; otherwise $L_i = \perp$. If $i = 1$ (i.e., there is no position on the left), then $L_i = *$. The meaning of the R_i 's is similar with respect to positions on the right of i .

A *profile Büchi automaton* \mathcal{A} is a Büchi automaton over the alphabet $\Sigma \times \{*, \top, \perp\} \times \{*, \top, \perp\}$. It defines a set $\mathcal{L}_{\text{data}}(\mathcal{A})$ of data words as follows: $w \in \mathcal{L}_{\text{data}}(\mathcal{A})$ if and only if \mathcal{A} accepts $\text{Profile}(w)$ in the standard sense.

A profile Büchi automaton with data-constraints is a tuple $(\mathcal{A}, \mathcal{C})$, where \mathcal{A} is a profile Büchi automaton and \mathcal{C} is a collection of data-constraints. It defines a set of data ω -words as follows. An data ω -word w is accepted by $(\mathcal{A}, \mathcal{C})$ if $\text{Profile}(w) \in \mathcal{L}(\mathcal{A})$ and $w \models \mathcal{C}$.

Theorem 5 *The emptiness problem for profile Büchi automata with data-constraints is in 2-NEXPTIME.*

We give a sketch of the proof in Subsection 4.2. The details can be found in Appendix D. Before that we give a slight extension of profile Büchi automata with data-constraints, which we call *profile Büchi automata with data-constraints on the state alphabet*. It is a trivial extension, but it will be very useful for our presentation in Appendix F.

4.1 Profile Büchi automata with data-constraints on the state alphabet

Definition 6 A profile Büchi automaton with *data-constraints on the state alphabet* is a pair $(\mathcal{A}, \mathcal{C})$, where

- $\mathcal{A} = \langle Q, q_0, \mu, F \rangle$ is a profile Büchi automaton, and
- \mathcal{C} is a collection of data-constraints over the *state* alphabet Q (instead of over the input alphabet Σ as in Definition 2).

Let $w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$ be an data ω -word, and $\rho = p_1 p_2 \cdots$ be a run of \mathcal{A} on $\text{Profile}(w)$. The *induced data word* of w on ρ is the data word $\rho(w) = \binom{p_1}{d_1} \binom{p_2}{d_2} \cdots$.

The automaton $(\mathcal{A}, \mathcal{C})$ accepts the data ω -word w , if there is an accepting run ρ of \mathcal{A} on $\text{Proj}(w)$ such that $\rho(w) \models \mathcal{C}$. We denote by $\mathcal{L}(\mathcal{A}, \mathcal{C})$ the language that consists of all the data ω -words accepted by the automaton $(\mathcal{A}, \mathcal{C})$.

The upper bound in Theorem 3 still holds for the emptiness problem of this type of automaton. Indeed given an input $(\mathcal{A}, \mathcal{C})$, a profile Büchi automaton with data-constraints on state alphabet Q , we can reduce it to $(\mathcal{A}', \mathcal{C}')$, a profile Büchi automaton with data-constraints over the alphabet $Q \times \Sigma$ as follows. The automaton \mathcal{A}' accepts the ω -word (with profiles) $(q_1, a_1, \text{profile}_1)(q_2, a_2, \text{profile}_2) \cdots$ if and only if $q_1 q_2 \cdots$ is an accepting run of the automaton \mathcal{A} on $(a_1, \text{profile}_1)(a_2, \text{profile}_2) \cdots$. The automaton \mathcal{A}' simply checks whether $(q_i, (a_i, \text{profile}_i), q_{i+1})$ is a valid transition in \mathcal{A} . Furthermore, the data-constraints over the alphabet Q can be reduced to data-constraints over the alphabet $(Q \times \Sigma)$ as follows.

1. The key-constraint $V(q) \mapsto q$ can be reduced to $V(q, a) \mapsto (q, a)$, for each $a \in \Sigma$ and denial-constraints $V(q, a) \cap V(q, b)$, whenever $a \neq b$ and $a, b \in \Sigma$.
2. The inclusion-constraint $V(q) \subseteq \bigcup_{p \in R} V(p)$ can be reduced to inclusion-constraints $V(q, a) \subseteq \bigcup_{p \in R, b \in \Sigma} V(p, b)$, for each $a \in \Sigma$.
3. The denial-constraint $V(q) \cap V(p) = \emptyset$ can be reduced to denial-constraints $V(q, a) \cap V(p, b) = \emptyset$, for each $a, b \in \Sigma$.

4.2 Sketch of proof of Theorem 5

The proof is an extension of the one in the previous section. However, we need a bit more auxiliary terms. Let $w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$ be an data ω -word over Σ . A *zone* is a maximal interval $[i, j]$ with the same data values, i.e. $d_i = d_{i+1} = \cdots = d_j$ and $d_{i-1} \neq d_i$ (if $i > 1$) and $d_j \neq d_{j+1}$ (if $j < n$). The zone $[i, j]$ is called an S -zone, if S is the set of labels occurring in the zone.

The *zonal partition* of w is a sequence (k_1, k_2, \dots) , where $1 \leq k_1 < k_2 < \cdots$ such that $[1, k_1], [k_1 + 1, k_2], \dots$ are the zones in w . Let the zone $[1, k_1]$ be an S_1 -zone, $[k_1 + 1, k_2]$ an S_2 -zone, $[k_2 + 1, k_3]$ an S_3 -zone, and so on. The *zonal word* of w is a data word over $\Sigma \cup 2^\Sigma$ defined as follows.

$$\text{Zonal}(w) = \binom{S_1}{d_{k_1}} a_1 \cdots a_{k_1} \binom{S_2}{d_{k_2}} a_{k_1+1} \cdots a_{k_2} \cdots$$

That is, the zonal word of a data word is a word in which each zone is preceded by a label $S \in 2^\Sigma$, if the zone is an S -zone.

Moreover, it is sufficient to assume that only the positions labeled with symbols from 2^Σ carry data values, i.e., the data values of their respective zones. Obviously each two consecutive zones have different data values, thus, two consecutive positions (in $\text{Zonal}(w)$) labeled with symbols from 2^Σ also have different data values.

Furthermore, if w is a data ω -word over Σ , then for each $a \in \Sigma$, $V_w(a) = \bigcup_{a \in S} V_{\text{Zonal}(w)}(S)$. Proposition 7 below shows that data-constraints for data words over the alphabet Σ can be converted into data-constraints for the zonal data words over the alphabet $\Sigma \cup 2^\Sigma$.

Proposition 7 *For every data word w over Σ , the following holds.*

- *A data ω -word w satisfies a key-constraint $V(a) \mapsto a$ if and only if its zonal data word $\text{Zonal}(w)$ satisfies the following constraints.*
 - K1. The key-constraints $V(R) \mapsto R$, for each R such that $a \in R$.*
 - K2. The denial-constraints $V(R) \cap V(R') \neq \emptyset$, for each R, R' such that $a \in R, R'$ and $R \neq R'$.*
 - K3. The symbol a occurs at most once in every zone in $\text{Zonal}(w)$.
(By a zone in $\text{Zonal}(w)$, we mean a maximal interval in which every positions are labeled with symbols from Σ .)*
- *A data ω -word w satisfies an inclusion-constraint $V(a) \mapsto \bigcup_{b \in S} V(b)$ if and only if its zonal data word $\text{Zonal}(w)$ satisfies the following inclusion-constraints:*

$$V(R) \subseteq \bigcup_{S' \cap S \neq \emptyset} V(S')$$

for each R such that $a \in R$.

- *A data ω -word w satisfies a denial-constraint $V(a) \cap V(b) = \emptyset$ if and only if its zonal data word $\text{Zonal}(w)$ satisfies the following denial-constraints:*

$$V(R) \cap V(R')$$

for each R and R' such that $a \in R$ and $b \in R'$.

Proof. The proof is straightforward due to the fact that

$$V_w(a) = \bigcup_{a \in S} V_{\text{Zonal}(w)}(S).$$

□

Now, given a profile automaton \mathcal{A} over the alphabet Σ , we can construct in exponential time an automaton $\mathcal{A}^{\text{ZONAL}}$ such that for all data ω -word w ,

$$\text{Profile}(w) \in \mathcal{L}(\mathcal{A}) \text{ if and only if } \text{Proj}(\text{Zonal}(w)) \in \mathcal{L}(\mathcal{A}^{\text{ZONAL}}).$$

Such an automaton $\mathcal{A}^{\text{ZONAL}}$ is called a *zonal automaton* of \mathcal{A} . Moreover, if the key-constraint $V(a) \mapsto a \in \mathcal{C}$, we can impose the condition *K3* in Proposition 7 inside the automaton $\mathcal{A}^{\text{ZONAL}}$. This, together with Proposition 7, implies that the emptiness problem of profile Büchi automata with data-constraints can be reduced to an instance of the following problem.

| | |
|-----------|---|
| PROBLEM: | OMEGA-SAT-ZONAL-AUTOMATA |
| INPUT: | <ul style="list-style-type: none"> • a zonal automaton $\mathcal{A}^{\text{ZONAL}}$ • a collection $\mathcal{C}^{\text{ZONAL}}$ of data-constraints over the alphabet 2^Σ |
| QUESTION: | is there a zonal word w such that <ul style="list-style-type: none"> • $\text{Proj}(w) \in \mathcal{L}(\mathcal{A}^{\text{ZONAL}})$ and $w \models \mathcal{C}^{\text{ZONAL}}$ and • in which two consecutive positions labeled with symbols from 2^Σ have different data values? |

The algorithm in Subsection 3.2 can be adapted to solve OMEGA-SAT-ZONAL-AUTOMATA. Extra cares are needed for the following two issues: (1) that each two consecutive zones must be assigned different data values, and (2) the possibility that the given zonal automaton accepts only ω -words with finitely many zones. We refer the reader to Appendix D for the details.

5 Two-variable logic for data ω -words

For the purpose of logical definability, we view data ω -words as structures

$$w = \langle \mathbb{N}, +1, \{a(\cdot)\}_{a \in \Sigma}, \sim \rangle, \quad (1)$$

where \mathbb{N} is the natural numbers $\{1, 2, \dots\}$ which indicates the positions, $+1$ is the successor relation (i.e., $+1(i, j)$ iff $i + 1 = j$), the $a(\cdot)$'s are the labeling predicates, and $i \sim j$ holds iff positions i and j have the same data value.

We let **FO** stand for first-order logic, **MSO** for monadic second-order logic (which extends **FO** with quantification over sets of positions), and **EMSO** for existential monadic second order logic, i.e., sentences of the form $\exists X_1 \dots \exists X_m \psi$, where ψ is an **FO** formula over the vocabulary extended with the unary predicates X_1, \dots, X_m . We let **FO**² stand for **FO** with two variables, i.e., the set of **FO** formulae that only use two variables x and

y . The set of all sentences of the form $\exists X_1 \dots \exists X_m \psi$, where ψ is an FO^2 formula is denoted by $\exists\text{MSO}^2$.

To emphasize that we are talking about a logic over data words we write $(+1, \sim)$ after the logic: e.g., $\text{FO}^2(+1, \sim)$ and $\exists\text{MSO}^2(+1, \sim)$. Note that $\exists\text{MSO}^2(+1)$ is equivalent in expressive power to MSO over the usual (not data) finite words, i.e., it defines precisely the regular languages [20].

It was shown in [3] that $\exists\text{MSO}^2(+1, <, \sim)$ is decidable over data words. In terms of complexity, the satisfiability of this logic is shown to be at least as hard as reachability in Petri nets. Without the $+1$ relation, the complexity drops to NEXPTIME -complete; however, without $+1$ the logic is not sufficiently expressive to capture regular relations on the data-free part of the finite word.

In this section we will prove the following:

Theorem 8 *The satisfiability problem is decidable for $\exists\text{MSO}^2(+1, \sim)$ over data ω -words. Moreover, the complexity of the decision procedure is elementary.*

5.1 A normal form for $\exists\text{MSO}^2(+1, \sim)$

Decidability proofs for two-variable logics typically follow this pattern: a syntactic normal form is established; to be followed by a combinatorial proof, where decidability is proved for that normal form (by establishing the finite-model property, or by automata techniques, for example).

Our proof is not different that it starts by establishing a normal form for $\text{FO}^2(+1, \sim)$, and then prove the decidability for the normal form. In fact, our normal form follows closely the one given in [2] for unranked finite data trees. It can simply be adapted it to the case of ω -words. It easily follows from [2] that every $\exists\text{MSO}^2(+1, \sim)$ sentence over data ω -words is equivalent to a sentence

$$\exists X_1 \dots \exists X_k (\chi \wedge \bigwedge_i \phi_i \wedge \bigwedge_j \psi_j)$$

where

1. χ is an $\text{FO}^2(+1)$ sentence over the extended alphabet $\Sigma \times \{*, \top, \perp\} \times \{*, \top, \perp\}$ (and it can be converted to a profile Büchi automaton in elementary complexity);
2. each ϕ_i is of the form $\forall x \forall y (\alpha(x) \wedge \alpha(y) \wedge x \sim y \rightarrow x = y)$, where α is a conjunction of labeling predicates, X_k 's, and their negations; and

3. each ψ_j is of the form $\forall x \exists y \alpha(x) \rightarrow (x \sim y \wedge \alpha'(y))$, with α, α' as in item 2.

The number of the unary predicates X 's is single exponential in the size of the original input sentence.

If we extend the alphabet to $\Sigma \times 2^k$ so that each label also specifies the family of the X_i 's the node belongs to, then sentences in items 2 and 3 can be encoded by data-constraints: formulae in item 2 become key- and denial-constraints, and formulae in item 3 become inclusion-constraints. Sentence (1) simply becomes an $\text{FO}^2(+1)$ sentence over the alphabet $\Sigma \times 2^k$.

Indeed, consider, for example, the sentence $\forall x \forall y (\alpha(x) \wedge \alpha(y) \wedge x \sim y \rightarrow x = y)$. Let Σ' be the set of all symbols $(a, \bar{b}) \in \Sigma \times 2^k$ consistent with α . That is, a is the labeling symbol used in α (if α uses one) or an arbitrary letter (if α does not use a labeling predicate), and the Boolean vector \bar{b} has 1 in positions of the X_i 's used positively in α and 0 in positions of X_j 's used negatively in α . Then the original sentence is equivalent to the key-constraints: $V(a) \mapsto a$, for each $a \in \Sigma'$ and denial-constraints: $V(a) \cap V(b) = \emptyset$, for every $a, b \in \Sigma'$ and $a \neq b$. The transformation of item 3 sentences into inclusion-constraints is the same.

Hence, the satisfiability problem of $\exists \text{MSO}^2(+1, \sim)$ can be reduced to the emptiness problem of profile Büchi automata with data-constraints, whose elementary complexity has been established in the previous section.

6 LTL that handles data values

In this section we extend the standard LTL with the operators $\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}$ to handle comparison between data values, which we denoted by $\text{LTL}[\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}]$.

Let Σ be a finite alphabet. Formally, the logic $\text{LTL}[\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}]$ is defined as follows.

- Both **True** and **False** are $\text{LTL}[\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}]$ formulae.
- For each $a \in \Sigma$, a is a $\text{LTL}[\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}]$ formula.
- If φ and ψ are $\text{LTL}[\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}]$ formulae, then so are

$$\neg \varphi \ ; \ \varphi \vee \psi \ ; \ \varphi \wedge \psi \ ; \ X \varphi \ ; \ \varphi \cup \psi \ ; \ \varphi \text{ R } \psi$$

- If φ is a $\text{LTL}[\Diamond^w, \Diamond^s, X_{\sim}, X_{\approx}]$ formula, then so are

$$\Diamond^w \varphi \ ; \ \Diamond^s \varphi \ ; \ X_{\sim} \varphi \ ; \ X_{\approx} \varphi$$

The operators X, U, R stand for neXt, Until and Release, respectively. We write $F\varphi$ as abbreviation for $\text{True}U\varphi$ and $G\varphi$ for $\neg F(\neg\varphi)$. The operators $\Diamond^w\varphi, \Diamond^s\varphi$ are to check the existence of a data value in the position where the formula φ holds.

We will not give the formal semantics of $\text{LTL}[\Diamond^w, \Diamond^s, X_\sim, X_\infty]$ here, which can be found in Appendix E. Instead we give only the intuitive meanings of the operators $\Diamond^w, \Diamond^s, X_\sim$ and X_∞ , which are as follows.

- The formula X_\sim holds in position i , if it has the same data value as the next position $i + 1$.
- The formula X_∞ holds in position i , if it has different data value as the next position $i + 1$.
- The formula $\Diamond^w\varphi$ holds in position i , if there exists a position j that has the same data value as position i and in which the formula φ holds.
- The formula $\Diamond^s\varphi$ holds in position i , if there exists a position $j \neq i$ that has the same data value as position i and in which the formula φ holds.

For an data ω -word w and a formula $\varphi \in \text{LTL}[\Diamond^w, \Diamond^s, X_\sim, X_\infty]$, we write $w, i \models \varphi$ to denote that in position i the formula φ holds. As usual, for a formula $\varphi \in \text{LTL}[\Diamond^w, \Diamond^s, X_\sim, X_\infty]$, we denote by $\mathcal{L}(\varphi)$ the set of words w for which $w, 1 \models \varphi$.

Notice the subtle difference between \Diamond^w and \Diamond^s , with w stands for “weak” and s for “strong,” respectively. With \Diamond^w it is not necessary that the position j is different from the current position, while with \Diamond^s the position j must be different. Obviously, \Diamond^w is weaker than \Diamond^s , as $\Diamond^w\varphi$ can be expressed as $\varphi \vee \Diamond^s\varphi$, hence the name “weak” and “strong.” In fact there exists a language expressible in $\text{LTL}[\Diamond^s]$, but not in $\text{LTL}[\Diamond^w]$.

At the first glance, it may appear that \Diamond^w is too weak to capture any interesting property. But as we will see later that the satisfiability problem even for $\text{LTL}[\Diamond^w]$ is already **NEXPTIME**-complete.

We will denote by $\text{LTL}[\Diamond^w]$ and $\text{LTL}[\Diamond^s]$ the class of formulae that uses only \Diamond^w and \Diamond^s , respectively, but do not use the operators X_\sim and X_∞ . We give some examples which will be used in the later sections.

Example 1 Consider the language $L_{\text{key}(a)}$ which consists of data words in which every two positions labeled with a have different data values. $L_{\text{key}(a)}$ is expressible by the $\text{LTL}[\Diamond^s]$ formula $G(a \rightarrow \neg\Diamond^sa)$. On the other hand,

the formula $\mathbf{G}(a \rightarrow \neg \Diamond^w a)$ does not make much sense as essentially it only expresses the data words in which the symbol a does not appear.

Example 2 Consider the formula $\varphi := \mathbf{G}(a \rightarrow \Diamond^s a)$ over the alphabet Σ . Then, $w \in \mathcal{L}(\varphi)$ if and only if every data value in $V_w(a)$ appears at least twice (among a -positions). This language $\mathcal{L}(\varphi)$ cannot be captured by an ADC.

Now consider a slightly different representation of the formula φ . Let $\bar{\Sigma}$ be a copy of the alphabet Σ , in which $\bar{b} \in \bar{\Sigma}$ denotes the corresponding symbol of $b \in \Sigma$. Consider the following formula $\varphi' := \mathbf{G}(a \rightarrow \Diamond^s \bar{a})$ over the alphabet $\Sigma \cup \bar{\Sigma}$. Essentially φ and φ' are equivalent up to renaming \bar{a} back to a . However, $\mathcal{L}(\varphi')$ can be captured by an ADC. This simple trick will be useful in our translation of $\text{LTL}[\Diamond^s]$ to an ADC for the purpose of deciding the satisfiability problem for $\text{LTL}[\Diamond^s]$.

Theorem 9

1. The satisfiability problem for $\text{LTL}[\Diamond^w]$ is *NEXPTIME*-complete.
2. The satisfiability problem for $\text{LTL}[\Diamond^s]$ is *2-NEXPTIME*.
3. The satisfiability problem for $\text{LTL}[\Diamond^s, X_{\sim}, X_{\approx}]$ is *3-NEXPTIME*.

The proofs for the upper bounds in Theorem 9 can be found in Appendix F. The proof for the hardness part in (1) can be found in Appendix G.

References

- [1] L. Boasson. Some applications of CFL's over infinite alphabets. *Theoretical Computer Science, LNCS vol. 104*, 1981, pages 146–151.
- [2] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on data trees and XML reasoning. *J. ACM* 56(3): (2009).
- [3] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, L. Segoufin. Two-variable logic on words with data. In *LICS'06*, pages 7-16.
- [4] P. Bouyer, A. Petit, D. Thérien. An algebraic characterization of data and timed languages. In *CONCUR'01*, pages 248–261.
- [5] E. Y. C. Cheng, M. Kaminski. Context-Free Languages over Infinite Alphabets. *Acta Inf.* 35(3): 245-267 (1998).
- [6] B. S. Chlebus. Domino-Tiling Games. *JCSS* 32(3): 374-392 (1986)

- [7] S. Dal-Zilio, D. Lugiez, C. Meyssonnier. A logic you can count on. In *POPL 2004*, pages 135–146.
- [8] C. David, L. Libkin, T. Tan. On the Satisfiability of Two-Variable Logic over Data Words. In *LPAR'10*, pages 248–262.
- [9] S. Demri, D. D'Souza, R. Gascon. A Decidable Temporal Logic of Repeating Values. In *LFCS'07*, pages 180–194.
- [10] S. Demri, R. Lazic. LTL with the freeze quantifier and register automata. *ACM TOCL* 10(3): (2009).
- [11] E. Grädel, Ph. Kolaitis, M. Vardi. On the decision problem for two-variable first-order logic. *BSL*, 3(1):53–69 (1997).
- [12] E. Grädel, M. Otto. On Logics with Two Variables. *TCS*, 224(1-2): 73–113 (1999).
- [13] M. Kaminski, N. Francez. Finite-memory automata. *TCS*, 134(2): 329–363 (1994).
- [14] M. Kaminski, T. Tan. Regular Expressions for Languages over Infinite Alphabets. *Fund. Inform.*, 69(3):301–318 (2006).
- [15] L. Libkin. Logics for Unranked Trees: An Overview. *Logical Methods in Computer Science* 2(3): (2006)
- [16] F. Neven. Automata, logic, and XML. In *CSL 2002*, pages 2–26.
- [17] F. Neven, Th. Schwentick, V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL* 5(3): (2004), 403–435.
- [18] Th. Schwentick. Automata for XML – a survey. *JCSS* 73 (2007), 289–315.
- [19] H. Seidl, Th. Schwentick, A. Muscholl, P. Habermehl. Counting in trees for free. In *ICALP 2004*, pages 1136–1149.
- [20] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages, Vol. 3*, Springer, 1997, pages 389–455.
- [21] P. Wolper. Constructing Automata from Temporal Logic Formulas: A Tutorial. In *European Educational Forum: School on Formal Methods and Performance Analysis*, Springer, 2000, pages 261–277.

A Analysis of the time complexity of the Algorithm in Subsection 3.2

Obviously Step (1) takes exponential time in the size of the alphabet Σ . Moreover, the sizes of the automaton $\tilde{\mathcal{A}}_{\text{fin}}$, the formula φ and the Büchi automaton $\tilde{\mathcal{A}}$ are all exponential in the size of the original alphabet Σ . The emptiness of Büchi automaton $\tilde{\mathcal{A}}$ can be checked in polynomial time, while the Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \mathcal{C})$ can be checked in NP. So overall our algorithm works in NEXPTIME.

B Proof of the correctness of the algorithm in Subsection 3.2

Throughout this section we fix an ADC $(\mathcal{A}, \mathcal{C})$ and $\mathcal{M} = \langle Q, \mu \rangle$ the transition system of \mathcal{A} , where $\mathcal{A} = \mathcal{M}_{q_0}^F$. We will demonstrate the following two claims, of which proofs are provided into the subsequent two subsections.

Claim 1 *Suppose there exists an data ω -word $w \in \mathcal{L}(\mathcal{A}, \mathcal{C})$. Then, by fixing $\mathcal{S}_0 = \mathcal{S}_0(w)$, $\mathcal{S}_{\text{fin}} = \mathcal{S}_{\text{fin}}(w)$ and $\mathcal{S}_{\infty} = \mathcal{S}_{\infty}(w)$, the constructed Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ and Büchi automaton $\tilde{\mathcal{A}}$ are both not empty.*

Claim 2 *Suppose there exist a partition $\mathcal{S}_0, \mathcal{S}_{\text{fin}}, \mathcal{S}_{\infty}$ of the set $2^Q - \{\emptyset\}$ such that the constructed Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ and Büchi automaton $\tilde{\mathcal{A}}$ are both not empty. Then, there exists an data ω -word $w \in \mathcal{L}(\mathcal{A}, \mathcal{C})$ such that $\mathcal{S}_0(w) = \mathcal{S}_0$, $\mathcal{S}_{\text{fin}}(w) = \mathcal{S}_{\text{fin}}$ and $\mathcal{S}_{\infty}(w) = \mathcal{S}_{\infty}$.*

We write $w[\leq i]$ to denote the initial segment of w of length i , while $w[\geq i]$ the ω -word obtained by discarding the initial segment of length $i - 1$ from w . Then, $\text{Proj}(w[\leq i]) = a_1 \cdots a_i$, and $\text{Proj}(w[\geq i]) = a_i a_{i+1} \cdots$.

B.1 Proof of Claim 1

Let w be an data ω -word accepted by $(\mathcal{A}, \mathcal{C})$. Let $\mathcal{S}_0 = \mathcal{S}_0(w)$, $\mathcal{S}_{\text{fin}} = \mathcal{S}_{\text{fin}}(w)$, and $\mathcal{S}_{\infty} = \mathcal{S}_{\infty}(w)$. Let N be the minimal index N such that for each $S \in \mathcal{S}_{\text{fin}}$, $[S]_{w[\leq N]} = [S]_w$.

Let $\rho = p_1 p_2 \cdots$ be the accepting run of \mathcal{A} on $\text{Proj}(w)$. Let $\tilde{\Sigma}$ and $\tilde{\mathcal{M}}$ be the new alphabet and the transition system constructed in Step (2) of our algorithm. Then, we pick the state p_N for the state q , supposedly be non-deterministically picked in Step (3) of our algorithm. The Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ constructed in Step (3) has the final state p_N , while the

Büchi automaton $\tilde{\mathcal{A}}$ has the initial state p_N . That is, $\tilde{\mathcal{A}}_{\text{fin}} = \tilde{\mathcal{M}}_{q_0}^{\{p_N\}}$ and $\tilde{\mathcal{A}} = \tilde{\mathcal{M}}_{p_N}^F$.

Consider the (without data) ω -word $x_1x_2\cdots$ over the alphabet $\tilde{\Sigma}$, where

$$x_i = \begin{cases} a_i \in \Sigma & \text{if } d_i \in [S]_w \text{ and } S \notin \mathcal{S}_\infty \\ (a_i, S) \in \Sigma_S & \text{if } d_i \in [S]_w \text{ and } S \in \mathcal{S}_\infty \end{cases}$$

We claim that the following words:

- $v_1 = x_1x_2\cdots x_N \in \mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$.
- $v_2 = x_{N+1}x_{N+2}\cdots \in \mathcal{L}(\tilde{\mathcal{A}})$.

B.1.1 Proof of $v_1 = x_1x_2\cdots x_N \in \mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$

There are two things to show here:

1. That v_1 is accepted by $\tilde{\mathcal{A}}_{\text{fin}}$.
2. That $\varphi(\text{Parikh}(x_1\cdots x_N))$ holds.

It is pretty straightforward to verify that $p_1\cdots p_N$ is a run of \mathcal{A} on $x_1\cdots x_N$. That it is an accepting run follows from the fact that q_N is a final state in $\tilde{\mathcal{A}}_{\text{fin}}$.

Now we will show that $\varphi(\text{Parikh}(q_1\cdots q_N))$ holds. Recall that the formula φ is of the form:

$$\exists z_{S_1} \cdots \exists z_{S_m} \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4$$

where

- the formula ψ_1 is the conjunction

$$\bigwedge_{a \in \Sigma} x_a \geq \sum_{S \ni a} z_S$$

- the formula ψ_2 is the conjunction

$$\bigwedge_{S \in \mathcal{S}_0 \cup \mathcal{S}_\infty} z_S = 0$$

- the formula ψ_3 is the conjunction

$$\bigwedge_{S \in \mathcal{S}_{\text{fin}}} z_S \geq 1$$

- the formula ψ_4 is the conjunction

$$\bigwedge_{V(a) \mapsto a \in \mathcal{C}} x_a = \sum_{a \in S} z_S$$

In order to show that $\varphi(\text{Parikh}(\text{Proj}(w)))$ holds, for each $S \subseteq Q$, we pick the following integers as witnesses for z_S .

- $z_S = |[S]_{w[\leq N]}|$, for each $S \in \mathcal{S}_{\text{fin}}(w)$.
- $z_S = 0$, for each $S \notin \mathcal{S}_{\text{fin}}(w)$.

We need to show that all the formulae ψ_1 – ψ_4 above are satisfied.

First, we observe that the following two points. For each $a \in \Sigma$,

1. $\#_a(v_1)$ is precisely the number of a -positions in $w[\leq N]$ whose data value is from the set

$$\bigcup_{S \in \mathcal{S}_{\text{fin}}} [S]_w$$

2. $\#_a(S)(v_1)$ is precisely the number of a -positions in $w[\leq N]$ whose data value is from the set $[S]_w$. Recall that in this case $S \in \mathcal{S}_{\infty}$.

Then, ψ_1 follows immediately from (1) that such $\#_a(v_1)$ number of a -positions must be greater than the number of its data values $\sum_{S \in \mathcal{S}_{\text{fin}}} |[S]_w|$. The formulae ψ_2 and ψ_3 follows immediately from the definition. That the formula ψ_4 holds is because of (1) and that the number $\#_a(v_1)$ of such a -positions is precisely the number of its data value $\sum_{S \in \mathcal{S}_{\text{fin}}} |[S]_w|$.

B.1.2 Proof of $v_2 = x_{N+1}x_{N+2} \cdots \in \mathcal{L}(\tilde{\mathcal{A}})$

Recall that the Büchi automaton $\tilde{\mathcal{A}}$ is the intersection of $\tilde{\mathcal{M}}_{p_N}^F$ with the automaton that checks the following condition.

1. Each $(a, S) \in \Sigma(\mathcal{S}_{\infty})$ appears infinitely many times.
2. If the key-constraint $V(a) \mapsto a \in \mathcal{C}$, then the symbol a does not appear.

Now, to show that $v_2 \in \mathcal{L}(\tilde{\mathcal{A}})$, we claim that $p_{N+1}p_{N+2} \cdots$ is also an accepting run of $\tilde{\mathcal{A}}$ on v_2 .

First, we show that $x_{N+1}x_{N+2} \cdots$ satisfies the properties (1) and (2) above. As $S \in \mathcal{S}_{\infty}(w)$, then it means each data values in $[S]_w$ appears infinitely many often in w . By our construction of $x_{N+1}x_{N+2} \cdots$, it means each symbol $(a, S) \in \Sigma(\mathcal{S}_{\infty})$ appears infinitely many often in $x_{N+1}x_{N+2} \cdots$.

Furthermore, recall that N is an index such that $[S]_{w[\leq N]} = [S]_w$, for each $S \in \mathcal{S}_{\text{fin}}(w)$. Now, if $w \models V(a) \mapsto a$, then every a -position greater than N in w has data value from the set $\bigcup_{S \in \mathcal{S}_{\infty}(w)} [S]_w$. This means that by our construction of $x_{N+1}x_{N+2}\dots$, the symbol a does not appear in $x_{N+1}x_{N+2}\dots$.

To show that $x_{N+1}x_{N+2}\dots$ is accepted by $\tilde{\mathcal{A}} = \tilde{\mathcal{M}}_{p_N}^F$, we observe that $p_{N+1}p_{N+2}\dots$ is an accepting run of $\tilde{\mathcal{A}}$ on $x_{N+1}x_{N+2}\dots$, which is immediate by our construction of \mathcal{M} .

B.2 Proof of Claim 2

Suppose there are the following items:

- $\mathcal{S}_0, \mathcal{S}_{\text{fin}}, \mathcal{S}_{\infty}$ is a partition of $2^{\Sigma} - \{\emptyset\}$;
- $\tilde{\Sigma} = \Sigma \cup \Sigma(\mathcal{S}_{\infty})$ and $\tilde{\mathcal{M}} = \langle \tilde{Q}, \tilde{\mu} \rangle$ be the constructed new alphabet and transition system;
- a state $q \in \tilde{Q}$,

such that the constructed the Presbruger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ and the Büchi automaton $\tilde{\mathcal{A}}$ are not empty. Consider the following two words.

- $v_1 = b_1 \dots b_N \in \mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$, where $p_1 \dots p_N$ be an accepting run of $\tilde{\mathcal{A}}_{\text{fin}}$ on v_1 .
- $v_2 = b_{N+1}b_{N+2}\dots \in \mathcal{L}(\tilde{\mathcal{A}})$, where $p_{N+1}p_{N+2}\dots$ be an accepting run of $\tilde{\mathcal{A}}$ on v_2 .

We will construct an data ω -word $w \in (\Sigma \times \mathfrak{D})^{\omega}$

$$w = \begin{pmatrix} a_1 \\ d_1 \end{pmatrix} \begin{pmatrix} a_2 \\ d_2 \end{pmatrix} \dots \begin{pmatrix} a_N \\ d_N \end{pmatrix} \begin{pmatrix} a_{N+1} \\ d_{N+1} \end{pmatrix} \dots,$$

which is accepted by $(\mathcal{A}, \mathcal{C})$.

We start by defining $\text{Proj}(w) = a_1a_2\dots$. For each $i = 1, 2, \dots$,

$$a_i = \begin{cases} b_i & \text{if } b_i \in \Sigma \\ c & \text{if } b_i = (c, S) \in \Sigma(\mathcal{S}_{\infty}) \text{ for some } S \end{cases}$$

By the construction of $\tilde{\mathcal{A}}_{\text{fin}}$ and $\tilde{\mathcal{A}}$, it is immediate that $p_1p_2\dots p_{N+1}p_{N+2}\dots$ is an accepting run of \mathcal{A} on $\text{Proj}(w)$.

Now we will define the data values d_1, d_2, \dots . For each $S \in \mathcal{S}_{\infty}$, we fix an infinite set of data values for \mathfrak{D}_S , such that all those sets \mathfrak{D}_S 's are disjoint. We will use \mathfrak{D}_S for $[S]_w$ for each $S \in \mathcal{S}_{\infty}$.

For each $S \in \mathcal{S}_{\text{fin}}$, the set $[S]_{\rho(w)}$ can be computed as follows. By the assumption, v_1 is a word such that $\varphi(\text{Parikh}(v_1))$ holds, where m_S is a witness for the variable z_S . Let $K = \sum_S m_S$. Define a function

$$\xi : \{1, \dots, K\} \rightarrow 2^\Sigma - \{\emptyset\},$$

such that $|\xi^{-1}(S)| = m_S$. We will use $\xi^{-1}(S)$ as $[S]_{\rho(w)} = \rho_{\rho(w)[\leq N]}$.

The assignment of data values to w can be done as follows.

1. We first define the data values for d_1, \dots, d_N . For each $a \in \Sigma$, pick the positions $Z(a) = \{i \mid b_i = a\}$. (Note that the parameter in defining the set $Z(a)$ of positions is the word $b_1 \dots b_N$.) Then we can assign those positions in $Z(a)$ with the data values from $\bigcup_{a \in S} \xi^{-1}(S)$. Such assignment is possible as $|Z(a)| = \#_{v_1}(a) \geq \sum_{a \in S} m_S$.
2. Then, we define the data values d_{N+1}, d_{N+2}, \dots , where $b_i \in \Sigma$. This is easy. We just pick some arbitrary data values from $\bigcup_{q \in S} \xi^{-1}(S)$.
3. At this stage we have define all the data values d_i 's for the positions i labeled with symbols from Σ in the $v_1 v_2$. What is left is to define the data values for the positions in $v_1 v_2$ whose labels are from $\Sigma(\mathcal{S}_\infty)$. Here we will use the data values in \mathfrak{D}_S and the assignment is done inductively. For each data value d in \mathfrak{D}_S that has not appeared yet in w , we pick $|S|$ number of positions $l_1, \dots, l_{|S|}$ in w such that $\{a_{l_1}, \dots, a_{l_{|S|}}\} = S$ and have no data values yet. Then, we assign all those positions with the data value d . By the acceptance criteria of the Büchi automaton $\tilde{\mathcal{A}}$, there are infinitely many such positions for each $S \in \mathcal{S}_\infty$. Thus, such assignment is always possible.

What remains now is to prove that $w \models \mathcal{C}$.

By Proposition 4 and the construction of \mathcal{S}_0 , as well as the Presburger formula φ , it is immediate that w satisfies the inclusion- and denial-constraints in \mathcal{C} . We will show that it also satisfies the key-constraints.

Suppose the key-constraint $V(a) \mapsto a \in \mathcal{C}$. First, in the assignment of data values in $w[\leq N]$ all the a -positions receive different data values, due to the constraint $|Z(a)| = \#_{v_1}(a) = \sum_{a \in S} m_S$. Second, from the construction of the automaton $\tilde{\mathcal{A}}$, the symbol a does not appear in $b_{N+1} b_{N+2} \dots$, thus not appearing in $w[\geq N+1]$. This means that we do not assign any data values from $\bigcup_{a \in S} \xi^{-1}(S)$ in every a -positions $\geq N+1$, so all data values in $\bigcup_{a \in S} \xi^{-1}(S)$ appears only in once in a -positions. Lastly, all the data values in $[S]_{\rho(w)}$ for each $S \in \mathcal{S}_\infty$ are assigned only once. Thus, it follows that every a -positions in w have different data values, thus, $w \models V(a) \mapsto a$. This completes the proof of Claim 2.

C The NP algorithm for Theorem 3

We identify that in our algorithm in Subsection 3.2, the exponential blow-up occurs in Step (1), where we have to enumerate all the non-empty subsets of Σ . Especially, the size of the set \mathcal{S}_∞ determines the sizes of the new alphabet $\tilde{\Sigma}$, the transition system $\tilde{\mathcal{M}}$. And the size of the set \mathcal{S}_{fin} determines the size of the Presburger formula φ .

The main idea of our NP is that if there is no key-constraint in \mathcal{C} , then the following holds. There exists a subset $\mathcal{Z} \subseteq 2^\Sigma$ of polynomial size such that there exists an data ω -word $w \in \mathcal{L}(\mathcal{A}, \mathcal{C})$ if and only if there exists an data ω -word $w' \in \mathcal{L}(\mathcal{A}, \mathcal{C})$, where $[S]_{w'} = \emptyset$, for all $S \notin \mathcal{Z}$. This means that in the constructions of $\tilde{\Sigma}$, $\tilde{\mathcal{M}}$, and φ , we only need to take into account the sets in \mathcal{Z} . This idea is the one that we are going to explain in the next subsection.

C.1 Preliminary notion

Let \mathcal{C} be a collection of inclusion- and denial-constraints. We define the subset $\mathcal{S}_0(\mathcal{C}) \subseteq 2^\Sigma$ as follows.

1. If \mathcal{C} contains the inclusion-constraint $V(a) \subseteq \bigcup_{b \in R} V(b)$, then $S \in \mathcal{S}_0(\mathcal{C})$ for all $S \subseteq \Sigma$ where $a \in S$ and $S \cap R = \emptyset$.
2. If \mathcal{C} contains the denial-constraint $V(a) \cap V(b) = \emptyset$, then $S \in \mathcal{S}_0(\mathcal{C})$ for all $S \subseteq \Sigma$ where S contains both a and b .

Remark 10 Given a non-empty set $S \subseteq \Sigma$, we can decide in polynomial time whether $S \in \mathcal{S}_0(\mathcal{C})$

C.2 The algorithm

Given an ADC $(\mathcal{A}, \mathcal{C})$, where \mathcal{C} does not contain key-constraints, the algorithm works as follows.

1. Construct (non-deterministically) a function $f : \Sigma \mapsto 2^\Sigma$ such that for each $a \in \Sigma$, either

$$a \in f(a) \text{ and } f(a) \notin \mathcal{S}_0(\mathcal{C})$$

or

$$f(a) = \emptyset$$

Such function can be non-deterministically constructed, by guessing $f(a)$ for each $a \in \Sigma$ and verify (in polynomial time) deterministically that $f(a) \notin \mathcal{S}_0(\mathcal{C})$.

2. Divide (non-deterministically) $\text{Image}(f)$ into two categories: \mathcal{S}_{fin} and \mathcal{S}_{∞} .

The intended meaning of \mathcal{S}_{fin} and \mathcal{S}_{∞} is the same as the algorithm in Subsection 3.2. Every other subsets not in $\mathcal{S}_{\text{fin}} \cup \mathcal{S}_{\infty}$ are considered in \mathcal{S}_0 .

3. Define the alphabet $\tilde{\Sigma} = \Sigma \cup \Sigma(\mathcal{S}_{\infty})$, where

$$\Sigma(\mathcal{S}_{\infty}) = \{(a, S) \mid a \in S \text{ and } S \in \mathcal{S}_{\infty}\},$$

and a transition system $\tilde{\mathcal{M}} = \langle \tilde{Q}, \tilde{\mu} \rangle$ over the alphabet $\tilde{\Sigma}$, as follows.

$$\begin{aligned} \tilde{Q} &= Q \\ \tilde{\mu} &= \mu \cup \{(p, (a, S), q) \mid (p, a, q) \in \mu \text{ and } (a, S) \in \Sigma(\mathcal{S}_{\infty})\} \end{aligned}$$

4. Non-deterministically choose one state $q \in \tilde{Q}$.
5. Construct a Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$, where $\tilde{\mathcal{A}}_{\text{fin}} = \tilde{\mathcal{M}}_{q_0}^{\{q\}}$ and the formula φ is as follows.
Let $\text{Image}(f) = \{S_1, \dots, S_l\}$. Then φ is of the form $\exists z_{S_1} \dots \exists z_{S_l} \psi$, where ψ is the following quantifier-free formula:

$$\bigwedge_{a \in \Sigma} \left(x_a \geq \sum_{S \ni a \text{ and } S \in \text{Image}(f)} z_S \right) \quad \wedge \quad \bigwedge_{S \in \mathcal{S}_{\infty}} z_S = 0 \quad \wedge \quad \bigwedge_{S \in \mathcal{S}_{\text{fin}}} z_S \geq 1$$

6. Construct a Büchi automaton $\tilde{\mathcal{A}}$ is simply the intersection of $\tilde{\mathcal{M}}_q^F$ with the automaton that checks that each $(a, S) \in \Sigma(\mathcal{S}_{\infty})$ appears infinitely many times.
7. Test the emptiness of $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ and $\mathcal{L}(\tilde{\mathcal{A}})$.
Then, $\mathcal{L}(\mathcal{A}, \mathcal{C}) \neq \emptyset$ if and only if $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi) \neq \emptyset$ and $\mathcal{L}(\tilde{\mathcal{A}}) \neq \emptyset$.

C.3 The proof of correctness

In view of Claims 1 and 2, to prove the correctness of our algorithm, it is sufficient to prove the following.

Claim 3 *If an data ω -word $w \models \mathcal{C}$, then there exist a function $f : \Sigma \mapsto 2^\Sigma$ that respects the condition in Step (1) and an data ω -word $v \models \mathcal{C}$ such that $\text{Proj}(v) = \text{Proj}(w)$ and $[S]_v = \emptyset$, for all $S \notin \text{Image}(f)$.*

Proof. Let

$$w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots .$$

We define the function f as follows. For each $a \in \Sigma$,

- if the label a does not appear in w , then $f(a) = \emptyset$;
- otherwise, define $f(a) = S_a$ such that $a \in S_a$ and $[S_a]_w \neq \emptyset$.
Such a set S_a exists as there is at least one a -position in w and this position has a data value in $V_w(a)$, which is partitioned into $\bigcup_{a \in S} [S]_w$.

We define the data word v as follows.

$$v = \binom{a_1}{d'_1} \binom{a_2}{d'_2} \cdots .$$

Thus, $\text{Proj}(v) = \text{Proj}(w)$. We define the data values d'_1, d'_2, \dots as follows.

- If $d_i \in [S]_w$, for some $S \in \text{Image}(f)$, then $d'_i = d_i$.
- If $d_i \notin [S]_w$, for all nonempty $S \in \text{Image}(f)$, then we pick arbitrary data value from $[f(a_i)]_w$ to assign to d'_i .

By such construction, we have $[S]_v = [S]_w$, for all non-empty $S \in \text{Image}(f)$. By Proposition 4, $v \models \mathcal{C}$. Furthermore, $[S]_v = \emptyset$, for all $S \notin \text{Image}(f)$. This completes the proof of our claim. \square

D Proof of Theorem 5

For the sake of presentation, we first show the decidability of a simpler version of the problem OMEGA-SAT-ZONAL-AUTOMATA, which we call OMEGA-SAT-LOCALLY-DIFFERENT in Subsection D.1. Then, in Subsection D.2 we explain how to adapt the approach in Subsection D.1 for OMEGA-SAT-ZONAL-AUTOMATA.

D.1 Locally different data ω -words

A data word $w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$ is called *locally different*, if each position has different data value from its left- and right-neighbors, that is, $d_i \neq d_{i+1}$, for each $i = 1, 2, \dots$.

In this section we give an algorithm to decide the problem SAT-LOCALLY-DIFFERENT defined below.

| | |
|-----------|--|
| PROBLEM: | OMEGA-SAT-LOCALLY-DIFFERENT |
| INPUT: | a Büchi automaton \mathcal{A} and a collection \mathcal{C} of key-, inclusion- and denial-constraints |
| QUESTION: | is there a locally different data word w such that $\text{Proj}(w) \in \mathcal{L}(\mathcal{A})$ with an accepting run ρ and $\rho(w) \models \mathcal{C}$? |

In the proof we will use the following simple lemma.

Lemma 11 [8, Lemma 3] *Let v be a finite data word over Σ . Suppose that for each $a \in \Sigma$, either $V_v(a) = \emptyset$ or $|V_v(a)| \geq |\Sigma| + 3$. Then we can rearrange the positions of the data values in v such that the resulting data word w is locally different, $\text{Proj}(w) = \text{Proj}(v)$ and for each $a \in \Sigma$, $V_w(a) = V_v(a)$.*

What this lemma tells us is that when the number of data values in found in a -positions is big enough, for each $a \in \Sigma$, then to solve SAT-LOCALLY-DIFFERENT, it is sufficient to solve OMEGA-SAT-ADC. Then, Lemma 11 allows us to rearrange the data values in the solution of OMEGA-SAT-ADC to be locally different.

In the rest of this section, the symbol ε denotes the constant $|\Sigma| + 3$. The main idea follows roughly as the one in the previous section, with the notable exception that for an data ω -word w , we divide the non-empty subsets $S \subseteq \Sigma$ into four categories:

- $\mathcal{S}_0(w) = \{S \mid [S]_w = \emptyset\}$.
- $\mathcal{S}_{\text{fin}}^{<\varepsilon}(w) = \{S \mid [S]_w \text{ is a finite set of cardinality } < \varepsilon\}$.
- $\mathcal{S}_{\text{fin}}^{\geq\varepsilon}(w) = \{S \mid [S]_w \text{ is a finite set of cardinality } \geq \varepsilon\}$.
- $\mathcal{S}_\infty(w) = \{S \mid [S]_w \text{ is an infinite set}\}$.

Note that in an data ω -word w , for $a \in S$ and $S \in \mathcal{S}_{\text{fin}}^{\geq\varepsilon}(w)$, then $V_w(a) \geq \varepsilon$. This will allow us to apply Lemma 11, for $V_w(a)$, where $a \in S$ and $S \in \mathcal{S}_{\text{fin}}^{\geq\varepsilon}(w)$. On the other hand, the data values in the sets $[S]_w$, where $S \in \mathcal{S}_{\text{fin}}^{<\varepsilon}(w)$ can be regarded as fixed constants, thus, can be embedded as

part of the input alphabet. This is our main idea to solve SAT-LOCALLY-DIFFERENT.

The details are as follows. Given an input $(\mathcal{A}, \mathcal{C})$, our algorithm does the following.

1. Guess a partition $\mathcal{S}_0, \mathcal{S}_{\text{fin}}^{<\varepsilon}, \mathcal{S}_{\text{fin}}^{\geq\varepsilon}, \mathcal{S}_\infty$ of the sets $2^Q - \{\emptyset\}$ as in the algorithm in Subsection 3.2.

That is, it respects the following conditions.

- C1. If the inclusion-constraint $V(a) \subseteq \bigcup_{b \in R} V(b)$ is in \mathcal{C} , then all the sets S , where $a \in S$ and $S \cap R = \emptyset$, are in \mathcal{S}_0 .
- C2. If the denial-constraint $V(a) \cap V(b) = \emptyset$ is in \mathcal{C} , then all the sets S , which contains both a and b , are in \mathcal{S}_0 .

2. Then, for each $S \in \mathcal{S}_{\text{fin}}^{<\varepsilon}$, we further guess a non-zero constant $K_S < \varepsilon$ and fix a set Γ_S of K_S number of constants. Define

$$\Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon}) = \{(a, d) \mid a \in S \text{ and } d \in \Gamma_S \text{ where } S \in \mathcal{S}_{\text{fin}}^{<\varepsilon}\}.$$

The intention is that we only need to consider the data ω -words w in which $[S]_w = \Gamma_S$, for each $S \in \mathcal{S}_{\text{fin}}^{<\varepsilon}$.

3. Let $\Sigma(\mathcal{S}_\infty) = \{(a, S) \mid a \in S \text{ and } S \in \mathcal{S}_\infty\}$. Construct the new alphabet $\hat{\Sigma}$, where

$$\hat{\Sigma} = \Sigma \cup \Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon}) \cup \Sigma(\mathcal{S}_\infty),$$

and the new transition system $\tilde{\mathcal{M}} = \langle \tilde{Q}, \tilde{\mu} \rangle$ is defined as:

$$\begin{aligned} \hat{Q} &= Q \\ \hat{\mu} &= \mu \cup \left\{ \begin{array}{l} \{(p, (a, S), q) \mid (p, a, q) \in \mu \text{ and } (a, S) \in \mathcal{S}(\infty)\} \\ \cup \\ \{(p, (a, d), q) \mid (p, a, q) \in \mu \text{ and } (a, d) \in \Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon})\} \end{array} \right\} \end{aligned}$$

4. Non-deterministically choose one state $q \in \tilde{Q}$.
5. Construct a Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ as follows.
 - (a) The automaton $\tilde{\mathcal{A}}_{\text{fin}}$ is $\mathcal{M}_{q_0}^{\{q\}}$ intersect with an automaton that checks the property:
 - If two symbols $(a, d_1), (b, d_2) \in \Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon})$ appear in two consecutive positions, then $d_1 \neq d_2$.

- If the key-constraint $V(a) \mapsto a \in \mathcal{C}$, then the symbol $(a, d) \in \Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon})$.
- (b) The Presburger formula φ is defined as follows. Let S_1, \dots, S_m be the enumeration of non-empty subsets of Q , where $m = 2^{|Q|} - 1$. The formula φ is of the form $\exists z_{S_1} \dots \exists z_{S_m} \psi$, where ψ is the following quantifier-free formula:

$$\begin{aligned}
& \bigwedge_{a \in \Sigma} \left(x_a \geq \sum_{S \ni a} z_S \right) \\
& \wedge \bigwedge_{S \in \mathcal{S}_0 \cup \mathcal{S}_{\text{fin}}^{<\varepsilon} \cup \mathcal{S}_{\infty}} z_S = 0 \quad \wedge \quad \bigwedge_{S \in \mathcal{S}_{\text{fin}}^{\geq \varepsilon}} z_S \geq 1 \\
& \wedge \bigwedge_{V(a) \mapsto a \in \mathcal{C}} \left(x_a = \sum_{a \in S} z_S \right)
\end{aligned}$$

6. Construct a Büchi automaton $\tilde{\mathcal{A}}$ as follows.
The Büchi automaton $\tilde{\mathcal{A}}$ is simply the intersection of $\tilde{\mathcal{M}}_q^F$ with the automaton that checks the following condition.
- (a) If two symbols $(a, d_1), (b, d_2) \in \Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon})$ appear in two consecutive positions, then $d_1 \neq d_2$.
 - (b) Each $(a, S) \in \Sigma(\mathcal{S}_{\infty})$ appears infinitely many times.
 - (c) If the key-constraint $V(a) \mapsto a \in \mathcal{C}$, then the symbols a and $(a, d) \in \Sigma \times \Gamma_S$, for some $S \in \mathcal{S}_{\text{fin}}^{<\varepsilon}$ do not appear.
7. Test the emptiness of $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ and $\mathcal{L}(\tilde{\mathcal{A}})$.
Then, $\mathcal{L}(\mathcal{A}, \mathcal{C}) \neq \emptyset$ if and only if $\mathcal{L}(\tilde{\mathcal{A}}_{\text{fin}}, \varphi) \neq \emptyset$ and $\mathcal{L}(\tilde{\mathcal{A}}) \neq \emptyset$.

The sizes of the automaton $\tilde{\mathcal{A}}_{\text{fin}}$, the formula φ and the Büchi automaton $\tilde{\mathcal{A}}$ are all exponential in the size of $(\mathcal{A}, \mathcal{C})$, thus, establishing the NEXPTIME upper bound for SAT-LOCALLY-DIFFERENT. The proof of correctness is similar to the proofs of the Claims 2 and 1. Lemma 11 ensures us that we get a locally different data ω -words. The constant data values from $\Sigma(\mathcal{S}_{\text{fin}}^{<\varepsilon})$ are already ensured by the automata $\tilde{\mathcal{A}}_{\text{fin}}$ and $\tilde{\mathcal{A}}$ that each of them does not appear in two consecutive positions.

D.2 The algorithm for OMEGA-SAT-ZONAL-AUTOMATA

Now we explain how the algorithm for OMEGA-SAT-LOCALLY-DIFFERENT can be adapted for OMEGA-SAT-ZONAL-AUTOMATA. It works as follows.

Given a zonal automaton \mathcal{A} and a collection \mathcal{C} of data-constraints over the alphabet 2^Σ , the algorithm does the following. It guesses if there exists a zonal word with infinitely many zones. If there is one, then the algorithm for OMEGA-SAT-LOCALLY-DIFFERENT can be adapted in a straightforward manner. Otherwise, it does the following. Let $\mathcal{M} = \langle Q, \mu \rangle$ be the transition system of \mathcal{A} , where $\mathcal{A} = \mathcal{M}_{q_0}^F$.

1. Guess a state $q \in Q$.
2. The presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$ is $\tilde{\mathcal{A}}_{\text{fin}} = \mathcal{M}_{q_0}^{\{q\}}$ over the alphabet $\Sigma \cup 2^\Sigma$ and the formula φ can be constructed like in Step (5) of the algorithm in Subsection D.1, but over the alphabet 2^Σ .
3. The Büchi automaton $\tilde{\mathcal{A}}$ is simply $\tilde{\mathcal{M}}_q^F$ intersects with an automaton that checks that the symbols from 2^Σ does not appear.

The intuition is that since there are only finitely many number zones, all the zones and its data-constraints are taken care by the Presburger automaton $(\tilde{\mathcal{A}}_{\text{fin}}, \varphi)$. The Büchi automaton $\tilde{\mathcal{A}}$ simply makes sure that the last zone has the property desired by the original Büchi automaton \mathcal{A} .

E The formal semantics of $\text{LTL}[\Diamond^w, \Diamond^s, \mathbf{X}_\sim, \mathbf{X}_\infty]$

Formally the semantics of $\text{LTL}[\Diamond^w, \Diamond^s, \mathbf{X}_\sim, \mathbf{X}_\infty]$ is given as follows. Let $w = \binom{a_1}{d_1} \binom{a_2}{d_2} \cdots$ and $i \in \{1, 2, \dots\}$.

- $w, i \models \text{True}$ and $w, i \not\models \text{False}$;
- $w, i \models a$ if and only if $a_i = a$;
- $w, i \models \varphi \vee \psi$ if and only if $w, i \models \varphi$ or $w, i \models \psi$;
- $w, i \models \neg\varphi$ if and only if $w, i \models \varphi$ is not true;
- $w, i \models \mathbf{X}\varphi$ if and only if $i + 1 \leq n$ and $w, i + 1 \models \varphi$;
- $w, i \models \mathbf{X}_\sim\varphi$ if and only if $i + 1 \leq n$ and $d_i = d_{i+1}$ and $w, i + 1 \models \varphi$;
- $w, i \models \mathbf{X}_\infty\varphi$ if and only if $i + 1 \leq n$ and $d_i \neq d_{i+1}$ and $w, i + 1 \models \varphi$;
- $w, i \models \varphi \mathbf{U} \psi$ if and only if there exists $j \geq i$ such that for all $i' = i, \dots, j - 1$, $w, i' \models \varphi$ and $w, j \models \psi$;

- $w, i \models \varphi \mathbf{R} \psi$ if and only if there exists $j \geq i$ such that $w, j \not\models \psi$, then there exists $i' \in \{i, \dots, j-1\}$, $w, i' \models \varphi$;
- $w, i \models \diamond^w \varphi$ if and only if there exists j such that $d_j = d_i$ and $w, j \models \varphi$;
- $w, i \models \diamond^s \varphi$ if and only if there exists $j \neq i$ such that $d_j = d_i$ and $w, j \models \varphi$.

A $\text{LTL}[\diamond^w, \diamond^s, \mathbf{X}_{\sim}, \mathbf{X}_{\infty}]$ formula φ defines a data language via $L(\varphi) = \{w \mid w, 1 \models \varphi\}$.

F Proofs of the upper bounds in Theorem 9

We first establish a normal form for formula in $\text{LTL}[\diamond^w, \diamond^s, \mathbf{X}_{\sim}, \mathbf{X}_{\infty}]$. A formula φ is in *normal form*, if every subformula in φ that starts with a negation, say $\neg\psi$, then ψ is either $a \in \Sigma$, or $\diamond^s \psi'$, or $\diamond^w \psi'$, for some ψ' .

Proposition 12 *Every formula φ in $\text{LTL}[\diamond^w, \diamond^s, \mathbf{X}_{\sim}, \mathbf{X}_{\infty}]$ can be converted to its equivalent normal form $\tilde{\varphi}$ in linear time.*

Proof. The construction $\tilde{\varphi}$ is done inductively.

- If φ does not start with a negation, then $\tilde{\varphi}$ is precisely φ .
- If φ is in the form $\neg\neg\psi$, then $\tilde{\varphi}$ is $\tilde{\psi}$.
- If φ is in the form $\neg(\psi \vee \psi')$, then $\tilde{\varphi}$ is $\widetilde{\neg\psi} \wedge \widetilde{\neg\psi'}$.
- If φ is in the form $\neg(\psi \wedge \psi')$, then $\tilde{\varphi}$ is $\widetilde{\neg\psi} \vee \widetilde{\neg\psi'}$.
- If φ is in the form $\neg\mathbf{X}\psi$, then $\tilde{\varphi}$ is $\mathbf{X}\widetilde{\neg\psi}$.
- If φ is in the form $\neg(\psi \mathbf{U} \psi')$, then $\tilde{\varphi}$ is $\widetilde{\neg\psi} \mathbf{R} \widetilde{\neg\psi'}$.
- If φ is in the form $\neg(\psi \mathbf{R} \psi')$, then $\tilde{\varphi}$ is $\widetilde{\neg\psi} \mathbf{U} \widetilde{\neg\psi'}$.
- If φ is in the form $\neg(\mathbf{X}_{\sim}\psi)$, then $\tilde{\varphi}$ is $(\mathbf{X}_{\infty}\mathbf{True}) \vee \mathbf{X}_{\sim}\widetilde{\neg\psi}$.
- If φ is in the form $\neg(\mathbf{X}_{\infty}\psi)$, then $\tilde{\varphi}$ is $(\mathbf{X}_{\sim}\mathbf{True}) \vee \mathbf{X}_{\infty}\widetilde{\neg\psi}$.
- If φ is in the form $\neg\diamond^w \psi$, then $\tilde{\varphi}$ is $\neg\diamond^w \tilde{\psi}$.
- If φ is in the form $\neg\diamond^s \psi$, then $\tilde{\varphi}$ is $\neg\diamond^s \tilde{\psi}$.

That φ and $\tilde{\varphi}$ are equivalent is straightforward. \square

Remark 13 It is straightforward from the construction of $\tilde{\varphi}$, that $\tilde{\varphi}$ stay in the same class as φ . That is,

- if $\varphi \in \text{LTL}[\Diamond^w]$, then $\tilde{\varphi} \in \text{LTL}[\Diamond^w]$;
- if $\varphi \in \text{LTL}[\Diamond^s]$, then $\tilde{\varphi} \in \text{LTL}[\Diamond^s]$; and
- if $\varphi \in \text{LTL}[\Diamond^w, \Diamond^s, \mathbf{X}_{\sim}, \mathbf{X}_{\infty}]$, then $\tilde{\varphi} \in \text{LTL}[\Diamond^w, \Diamond^s, \mathbf{X}_{\sim}, \mathbf{X}_{\infty}]$.

F.1 The NEXPTIME upper bound for part (1) of Theorem 9

By Proposition 12, we can assume that the input formula is always in normal form. The proof of decidability itself is done by translating the input formula $\varphi \in \text{LTL}[\Diamond^w]$ to an equivalent ADC $(\mathcal{A}_{\varphi}, \mathcal{C}_{\varphi})$. The translation follows closely the classical translation from standard LTL to Büchi automaton. (See, for example, [21].) So we simply sketch it here. We recall the standard notion of the closure of the formula φ , denoted by $\text{Cl}(\varphi)$.

- $\varphi \in \text{Cl}(\varphi)$.
- $a \in \text{Cl}(\varphi)$, for each $a \in \Sigma$.
- If $\neg a \in \text{Cl}(\varphi)$, then $\bigvee_{b \in \Sigma - \{a\}} b$.
- If $\varphi_1 \wedge \varphi_2 \in \text{Cl}(\varphi)$, then $\varphi_1, \varphi_2 \in \text{Cl}(\varphi)$.
- If $\varphi_1 \vee \varphi_2 \in \text{Cl}(\varphi)$, then $\varphi_1, \varphi_2 \in \text{Cl}(\varphi)$.
- If $\mathbf{X} \varphi_1 \in \text{Cl}(\varphi)$, then $\varphi_1 \in \text{Cl}(\varphi)$.
- If $\varphi_1 \mathbf{U} \varphi_2 \in \text{Cl}(\varphi)$, then $\varphi_1, \varphi_2 \in \text{Cl}(\varphi)$.
- If $\varphi_1 \mathbf{R} \varphi_2 \in \text{Cl}(\varphi)$, then $\varphi_1, \varphi_2 \in \text{Cl}(\varphi)$.
- If $\Diamond^w \varphi_1 \in \text{Cl}(\varphi)$, then $\varphi_1 \in \text{Cl}(\varphi)$.
- If $\neg \Diamond^w \varphi_1 \in \text{Cl}(\varphi)$, then $\varphi_1 \in \text{Cl}(\varphi)$.

The standard construction of $\mathcal{A}_{\varphi} = \langle Q, q_0, \mu, F \rangle$ will yield $Q \subseteq 2^{\text{Cl}(\varphi)}$, where $q \in Q$ if the conditions hold.

(S1) $\text{False} \notin q$;

- (S2) $q \cap \Sigma$ is a singleton;
- (S3) if $\varphi_1 \in q$, then the normal form $\neg\overline{\varphi_1} \notin q$;
- (S4) if the normal form $\neg\overline{\varphi_1} \in q$, then $\varphi_1 \notin q$;
- (S5) if $\varphi_1 \wedge \varphi_2 \in q$, then $\varphi_1, \varphi_2 \in q$;
- (S6) if $\varphi_1 \vee \varphi_2 \in q$, then $\varphi_1 \in q$ or $\varphi_2 \in q$.

Intuitively, the meaning of \mathcal{A}_φ is such that in every state q , it takes care that every formula $q - \{\diamond^w \psi \mid \psi \in \text{Cl}(\varphi)\}$ holds. The construction of q_0 , μ and F are standard like in [21], thus, omitted. The set \mathcal{C} of constraints will take care of the operator \diamond^w . It consists of the following.

1. For every state q that contains the sub-formula $\diamond^w \psi$, then \mathcal{C}_φ contains the constraints:

$$V(q) \subseteq \bigvee_{\psi \in q'} V(q').$$

2. For every state q that contains the sub-formula $\neg\diamond^w \psi$, then \mathcal{C}_φ contains the constraints:

$$V(q) \cap V(q') = \emptyset,$$

for all q' that contains ψ .

Now \mathcal{C} does not contain key-constraints. The construction of \mathcal{A}_φ is already in EXPTIME. By NP upper bound in Theorem 3, we get the NEXPTIME upper bound for the satisfiability problem of $\text{LTL}[\diamond^w]$.

F.2 The 2-NEXPTIME upper bound for part (2) of Theorem 9

By Proposition 12, we assume that the input formula φ is in normal form. Again, the proof of decidability is done by translating the input formula $\varphi \in \text{LTL}[\diamond^s]$ to an equivalent ADC $(\mathcal{A}_\varphi, \mathcal{C}_\varphi)$. However, we have to make a bit of modification because, as explained in Example 2, formulas such as $\mathbf{G}(a \rightarrow \diamond^s a)$ cannot be directly translated to an ADC.

We apply the same trick as in Example 2 to make a copy $\overline{\psi}$ of each formula ψ in $\text{Cl}(\varphi)$. The idea is that the copy $\overline{\psi}$ has exactly the same property as the formula ψ . We denote by $\overline{\text{Cl}(\varphi)}$ the set of all such copies.

Then the ADC $(\mathcal{A}_\varphi, \mathcal{C}_\varphi)$ is defined over the alphabet $\Sigma \cup \overline{\Sigma}$ as follows. The automaton $\mathcal{A}_\varphi = \langle Q, q_0, \mu, F \rangle$ is such that $Q \subseteq 2^{\text{Cl}(\varphi) \cup \overline{\text{Cl}(\varphi)}}$. A state $q \in Q$ if in addition to Conditions (S1)–(S6) above, the following conditions hold.

- both False and $\overline{\text{False}}$ are not in q ;
- $q \cap (\Sigma \cup \overline{\Sigma})$ is a singleton;
- if $\varphi_1 \in q$, then both $\neg\varphi_1$ and $\overline{\neg\varphi_1}$ are not in q ;
- if $\neg\varphi_1 \in q$, then both φ_1 and $\overline{\varphi_1}$ are not in q ;
- if $\varphi_1 \wedge \varphi_2 \in q$, then either
 - $\varphi_1, \varphi_2 \in q$, or
 - $\overline{\varphi_1}, \varphi_2 \in q$, or
 - $\varphi_1, \overline{\varphi_2} \in q$, or
 - $\overline{\varphi_1}, \overline{\varphi_2} \in q$;
- if $\overline{\varphi_1 \wedge \varphi_2} \in q$, then either
 - $\varphi_1, \varphi_2 \in q$, or
 - $\overline{\varphi_1}, \varphi_2 \in q$, or
 - $\varphi_1, \overline{\varphi_2} \in q$, or
 - $\overline{\varphi_1}, \overline{\varphi_2} \in q$;
- if $\varphi_1 \vee \varphi_2 \in q$, then one of $\varphi_1, \varphi_2, \overline{\varphi_1}, \overline{\varphi_2} \in q$;
- if $\overline{\varphi_1 \vee \varphi_2} \in q$, then one of $\varphi_1, \varphi_2, \overline{\varphi_1}, \overline{\varphi_2} \in q$;
- if $\overline{\varphi_1} \in q$, then $\varphi_1 \notin q$;
- if $\varphi_1 \in q$, then $\overline{\varphi_1} \in q$;
- if $\overline{\Diamond^s \varphi_1} \in q$, then either $\Diamond^s \varphi \in q$, or $\Diamond^s \overline{\varphi_1} \in q$;
- if φ_1 , then $\Diamond^s \varphi_1 \notin q$;
- if $\overline{\varphi_1}$, then $\Diamond^s \overline{\varphi_1} \notin q$.

Note that in such construction the states that are supposed to contain both φ_1 and $\Diamond^s \varphi_1$ are replaced by states that contain either

- both φ_1 and $\Diamond^s \overline{\varphi_1}$, or
- both $\overline{\varphi_1}$ and $\Diamond^s \varphi$.

The construction of q_0 , μ and F is standard.

The collection \mathcal{C}_φ of data-constraints consists of the following.

1. For each state q that contains both the sub-formulae ψ and $\neg\Diamond^s\psi$, \mathcal{C}_φ contains:
 - the key-constraints $V(q) \mapsto q$; and
 - the denial-constraints $V(q) \cap V(p) = \emptyset$, for all state p that contains ψ .

The same if q contains both $\overline{\psi}$ and $\neg\Diamond^s\overline{\psi}$

2. For each state q that contains the sub-formula $\Diamond^s\psi$ but not the sub-formula ψ , \mathcal{C}_φ contains the inclusion-constraints

$$V(q) \subseteq \bigvee_{\psi \in p} V(p).$$

3. For each state q that contains the sub-formula $\neg\Diamond^s\psi$ but not the sub-formula ψ , \mathcal{C}_φ contains the denial-constraints

$$V(q) \cap V(p) = \emptyset,$$

for every state p that contains ψ .

The construction of \mathcal{A}_φ is already in NEXPTIME. By Theorem 3, we get the 2-NEXPTIME upper bound for $\text{LTL}[\Diamond^s]$.

F.3 The 3-NEXPTIME upper bound for part (3) of Theorem 9

If we have the local comparison \mathbf{X}_\sim and \mathbf{X}_∞ , it can be handled with the addition of profile in the automata. As the inclusion of profile constraints induce an exponential blow-up, we get 3-NEXPTIME upper bound. The construction is straightforward, thus, omitted.

G The NEXPTIME-hardness of $\text{LTL}[\Diamond]$

In the proof of the following theorem we will use $\Box^w\varphi$ as an abbreviation of $\neg\Diamond^w\neg\varphi$.

Theorem 14 *The satisfiability problem for $\text{LTL}[\Diamond^w]$ on (finite and infinite) data words is NEXPTIME-hard.*

Proof. The proof is by reduction from the 2^n -corridor tiling problem. An instance $I = (T, H, V, F, L, n)$ of this problem consists of a finite set T of tile types, horizontal and vertical constraints $H, V \subseteq T \times T$, constraints $F, L \subseteq T$ for the first and last row and a number n given in unary. The task is to decide, whether T tiles the $2^n \times 2^n$ -corridor, respecting the constraints. This problem is NEXPTIME-hard [6].

For an arbitrary instance $I = (T, H, V, F, L, n)$ of the 2^n -corridor tiling problem we will construct a formula φ_I of polynomial length (in $|I|$) which is satisfiable if and only if I has a solution.

We use $\Sigma = T \cup \{0, 1\} \cup \{c, r, u\}$ as the underlying alphabet. The idea is to assign to every square on the tiling grid a column and a row number to be able to check the constraints. We use data values as pointers to the binary encoding of a number. We first introduce some abbreviations. A bit is represented by two successive positions in the data word. The first one is labelled by 0 or 1 and the data value of the second position serves as a pointer to the position with the next bit. It is crucial that all positions pointed by the same pointer carry the same bit value. The following formula ensures that this property holds. Since we will encode binary numbers with n bits, the X -operator is used $n - 1$ times.

$$\begin{aligned} \varphi_{bitstring} = & \Diamond^w((0 \vee 1) \wedge X\Diamond^w((0 \vee 1) \wedge X\Diamond^w(\dots \wedge X\Diamond^w(0 \vee 1))\dots)) \wedge \\ & (\Box^w 0 \vee \Box^w 1) \wedge \Box^w X((\Box^w 0 \vee \Box^w 1) \wedge \Box^w X((\Box^w 0 \vee \Box^w 1) \wedge \dots \wedge \Box^w X(\Box^w 0 \vee \Box^w 1)\dots)) \end{aligned}$$

The next formula encodes the number 0 in binary.

$$\varphi_0 = \Box^w(0 \wedge X\Box^w(0 \wedge X\Box^w(\dots \wedge X\Box^w 0)\dots))$$

The following encodes number $2^n - 1$.

$$\varphi_1 = \Box^w(1 \wedge X\Box^w(1 \wedge X\Box^w(\dots \wedge X\Box^w 1)\dots))$$

The next formula says that the i th bit encodes bit value b . The expression $(X\Box^w)^{i-1}$ means that $X\Box^w$ is repeated $i - 1$ times.

$$bit-i-b = \Box^w(X\Box^w)^{i-1}b$$

for $1 \leq i \leq n$ and $b \in \{0, 1\}$.

It should be noted that the first bit serves as the lowest bit.

The formula φ_I is composed of the formulas ψ and χ : the formula ψ describes the encoding of the tiling grid and χ describes the constraints which has to hold.

Every square of the tiling is represented by a sequence of four positions in the data word. The first position is labeled by the tile type belonging to this square, the second one serves as a pointer to the bit representation of the column number of the square, the third one serves as a pointer to the bit representation of the row number of the square and the fourth one serves as an *up-pointer* to the next upper square on the same column. Such a sequence of positions will be called *square encoding*.

$$\begin{aligned}\psi_1 &= G[\bigvee_{t \in T} t \rightarrow \mathbf{X}(c \wedge \varphi_{bitstring})] \\ \psi_2 &= G[\bigvee_{t \in T} t \rightarrow \mathbf{XX}(r \wedge \varphi_{bitstring})] \\ \psi_3 &= G[\bigvee_{t \in T} t \rightarrow \mathbf{XXX}(u \wedge \square^w \bigvee_{t \in T} t)]\end{aligned}$$

The first $4 \cdot 2^{2n}$ positions of the word represent a list of all square encodings of the tiling. The list begins with the square with column number 0 and row number 0. After all 2^n squares of a row are listed the first square of the next row follows.

First we have to ensure that the first square encoding has row number 0 and column number 0.

$$\psi_4 = \mathbf{X}\varphi_0 \wedge \mathbf{XX}\varphi_0$$

A square encoding with column number $i < 2^n - 1$ and row number j is followed by a square encoding with column number $i + 1$ and row number j .

$$\begin{aligned}\psi_5 &= G[(\bigvee_{t \in T} t \wedge \mathbf{X}\neg\varphi_1) \rightarrow (\bigwedge_{i=1}^n ((\mathbf{XX}bit-i-0 \rightarrow \mathbf{XXXX}(\bigvee_{t \in T} t \wedge \mathbf{XX}bit-i-0)) \\ &\quad \wedge (\mathbf{XX}bit-i-1 \rightarrow \mathbf{XXXX}(\bigvee_{t \in T} t \wedge \mathbf{XX}bit-i-1)))))] \\ \psi_6 &= G[(\bigvee_{t \in T} t \wedge \mathbf{X}\neg\varphi_1) \rightarrow \bigvee_{i=1}^n (\bigwedge_{j=1}^{i-1} (\mathbf{X}bit-j-1 \wedge \mathbf{XXXXX}bit-j-0) \wedge \\ &\quad \mathbf{X}bit-i-0 \wedge \mathbf{XXXXX}bit-i-1 \wedge \\ &\quad \bigwedge_{j=i+1}^n ((\mathbf{X}bit-j-0 \rightarrow \mathbf{XXXXX}bit-j-0) \wedge (\mathbf{X}bit-j-1 \rightarrow \mathbf{XXXXX}bit-j-1)))]\end{aligned}$$

A square encoding with column number $2^n - 1$ and row number $j < 2^n - 1$ is followed by a square encoding with column number 0 and row number $j + 1$.

$$\begin{aligned}\psi_7 = & G[(\bigvee_{t \in T} t \wedge \mathbf{X}\varphi_1 \wedge \mathbf{X}\neg\varphi_1) \rightarrow (\mathbf{XXXX}(\bigvee_{t \in T} t \wedge \mathbf{X}\varphi_0))] \\ \psi_8 = & G[(\bigvee_{t \in T} t \wedge \mathbf{X}\varphi_1 \wedge \mathbf{X}\neg\varphi_1) \rightarrow \bigvee_{i=1}^n (\bigwedge_{j=1}^{i-1} (\mathbf{XX}bit-j-1 \wedge \mathbf{XXXXXX}bit-j-0) \wedge \\ & \mathbf{XX}bit-i-0 \wedge \mathbf{XXXXXX}bit-i-1 \wedge \\ & \bigwedge_{j=i+1}^n ((\mathbf{XX}bit-j-0 \rightarrow \mathbf{XXXXXX}bit-j-0) \wedge (\mathbf{XX}bit-j-1 \rightarrow \mathbf{XXXXXX}bit-j-1)))]\end{aligned}$$

After the square encoding with column number $2^n - 1$ and row number $2^n - 1$ there follow no more positions labelled with a tile type. By this we ensure that every square encoding occurs exactly once.

$$\psi_9 = G[(\bigvee_{t \in T} t \wedge \mathbf{X}\varphi_1 \wedge \mathbf{X}\mathbf{X}\varphi_1) \rightarrow \mathbf{X}G \bigwedge_{t \in T} \neg t]$$

The up-pointer of every square encoding with column number i and row number $j < 2^n - 1$ points to the first position of the unique square encoding with column number i and row number $j + 1$.

$$\begin{aligned}\psi_{10} = & G[(\bigvee_{t \in T} t \wedge \mathbf{X}\neg\varphi_1) \rightarrow (\bigwedge_{i=1}^n ((\mathbf{X}bit-i-0 \rightarrow \mathbf{XXX}\Box^w(\bigvee_{t \in T} t \wedge \mathbf{X}bit-i-0) \wedge \\ & (\mathbf{X}bit-i-1 \rightarrow \mathbf{XXX}\Box^w(\bigvee_{t \in T} t \wedge \mathbf{X}bit-i-1)))))] \\ \psi_{11} = & G[(\bigvee_{t \in T} t \wedge \mathbf{X}\neg\varphi_1) \rightarrow \bigvee_{i=1}^n (\bigwedge_{j=1}^{i-1} (\mathbf{XX}bit-j-1 \wedge \mathbf{XXX}\Box^w\mathbf{XX}bit-j-0) \wedge \\ & \mathbf{XX}bit-i-0 \wedge \mathbf{XXX}\Box^w\mathbf{XX}bit-i-1 \wedge \\ & \bigwedge_{j=i+1}^n ((\mathbf{XX}bit-j-0 \rightarrow \mathbf{XXX}\Box^w\mathbf{XX}bit-j-0) \wedge (\mathbf{XX}bit-j-1 \rightarrow \mathbf{XXX}\Box^w\mathbf{XX}bit-j-1)))]\end{aligned}$$

The following formulas express that the constraints in I are respected. The squares of the first row carry only tile types from F .

$$\chi_1 = G[(\bigvee_{t \in T} t \wedge \mathbf{X}\mathbf{X}\varphi_0) \rightarrow \bigvee_{t \in F} t]$$

Similarly, the squares of the last row carry only tile types from L .

$$\chi_2 = G[(\bigvee_{t \in T} t \wedge \mathbf{xx}\varphi_1) \rightarrow \bigvee_{t \in L} t]$$

The tile type of a square and the tile type of his right neighbor respect the horizontal constraints.

$$\chi_3 = G[\bigwedge_{t \in T} ((t \wedge \mathbf{x}\neg\varphi_1) \rightarrow \mathbf{xxxx} \bigvee_{(t,t') \in H} t')]$$

The tile type of a square and the tile type of his upper neighbor respect the vertical constraints.

$$\chi_4 = G[\bigwedge_{t \in T} ((t \wedge \mathbf{xx}\neg\varphi_1) \rightarrow \mathbf{xxx}\square^w \bigvee_{(t,t') \in V} t')]$$

The desired formula is $\varphi_I = \bigwedge_{i=1}^{11} \psi_i \wedge \bigwedge_{j=1}^4 \chi_j$. It's easy to see that φ_I is satisfiable if and only if I has a solution. \square